

---

# 1. Introduction

## 1.1. About Maximus

Maximus is a flexible bulletin board package for the DOS and OS/2 operating systems. Maximus allows remote users to connect to the system by modem, read and write messages, participate in public conference areas, send and receive files, and much more.

In addition to the standard message and file features found in most BBS programs, Maximus also includes:

- MEX, an extension language that combines the best elements of the C, Pascal and BASIC languages. MEX includes support for advanced language features such as structures, arrays, dynamic strings, and pass-by-reference function arguments. MEX can be used to customize and extend Maximus in an infinite number of ways.
- MECCA, an easy-to-use scripting language that can be used to colorize screens and add simple menus and prompts to display files.
- Support for *RIPscrip* graphics. Maximus can detect *RIPscrip* capabilities, automatically size menu output based on the terminal window size, display most internal prompts using *RIPscrip* graphics, and much more.
- Support for SM, a Presentation Manager LAN monitoring tool for OS/2. SM can be used to manipulate and examine multiple Maximus sessions running on remote workstations.
- Full support for CD-ROMs and other slow filesystems. Maximus will copy files to a staging area before a transfer and it will only access the drive when absolutely necessary. Areas can also be specifically excluded from new files searches.
- A message tracking system for use in technical support environments. Maximus can keep an audit trail of all messages in certain areas, assign “ownership” of messages to individuals, and produce detailed reports regarding the status of various messages.
- Support for an unlimited number of message and file areas. Maximus also supports “divisions” for constructing multi-level message and file area hierarchies.

- Support for a REXX user file interface. This interface can be used to read from and write to the Maximus user database from any OS/2-based REXX program.
- A powerful message “browse” feature that allows selection of messages based on user-defined search criteria.
- A built-in QWK mail subsystem that allows users to read and compose messages while off-line.
- Support for the Squish message format, a compact database for storing messages.
- Optional support for user password encryption. This feature uses the RSA Data Security, Inc. Message-Digest 5 algorithm as a one-way hash for storing passwords in the user file.
- Full multilingual support. The Maximus language file can be customized to support almost any language. Maximus also includes special support for the Swedish 7-bit and the Chinese BIG5 character sets.
- An internal multinode chat facility, including virtual CB channels and private chatting between two nodes.
- An internal full-screen editor for composing messages.
- Internal file transfer protocols, including Zmodem, Ymodem-G and other popular protocols.

## 1.2. System Requirements

This chapter describes the minimum hardware requirements for a standard Maximus installation.

### 1.2.1. MS-DOS or PC-DOS Requirements

The minimum system requirements for the DOS version of Maximus are:

- an IBM (or 100% compatible) personal computer with at least 450K of available conventional RAM,
- MS-DOS or PC-DOS version 3.3 or greater, and
- A FOSSIL communications driver (revision level 5 or higher). (Common FOSSIL drivers are X00, BNU and OpusComm.)

### 1.2.2. OS/2 Requirements

The minimum system requirements for the OS/2 version of Maximus are:

- an IBM (or 100% compatible) personal computer with at least four megabytes of installed memory, and
- one of the following products:
  - ◆ IBM OS/2, Version 2.0, 2.1 or 2.11
  - ◆ IBM OS/2 Warp, Version 3 or above
  - ◆ IBM OS/2 Warp Connect, Version 3 or above

### 1.2.3. Common Requirements

Regardless of the operating system type, the following hardware is also required to run Maximus:

- A Hayes-compatible modem, 1200 bps or faster.
- A hard disk with at least 15 megabytes of free space. Additional space is required to store the contents of file and message areas.

## 1.3. Typographical Conventions

The following typographical conventions are used in this manual:

Commentary, descriptions, and general discussions are set in this typeface.

Source code, batch files, and command line examples are set in this typeface.

MECCA tokens, MEX data types and MEX variable names are *italicized* in discussion text.

Filenames, menu options, MEX function names, MEX keywords, and characters entered by the user are set in **boldface**.



---

## 2. Maximus Overview

This chapter provides a brief list of the commands and features supported by Maximus 3.0. This list is by no means complete, but it serves as a useful introduction to those System Operators (SysOps) who are unfamiliar with Maximus.

### 2.1. Waiting for Callers

When Maximus is set up to handle remote callers, it enters “Waiting for Caller” (WFC) mode as soon as the system is started. In this mode, Maximus initializes the modem and instructs it to wait for incoming calls.

When a ring is detected, the modem answers the phone. If a connection is successfully established, Maximus waits until the user presses `<enter>` twice, or until five seconds have elapsed, whichever occurs sooner.

### 2.2. Logging On

After a connection is established, Maximus displays the system name and version, followed by any information that the SysOp has placed in the log-on display file, `\max\misc\logo.mec`. This screen must not include any graphics commands or high-bit characters, since Maximus has not yet determined the graphics capabilities of the caller.

Maximus then prompts the user for a name. Unlike other BBS programs, Maximus allows more than two words in a username, so names such as “John Q. Public” are perfectly acceptable. However, Maximus rejects callers with one-word names unless the **Single Word Names** feature is enabled.

If the user does not exist in the user file, Maximus displays the `\max\misc\notfound.mec` file. This file normally informs the user that no existing user record with the specified name could be found, and it normally indicates that a new account will be created if the user continues with the log-on process.

Next, Maximus displays a prompt to ensure that the user’s name was entered correctly. Given input of “John Doe”, Maximus will respond with:

```
John Doe [Y,n]?
```

If the username was spelled incorrectly, the user can enter “n” and begin the log-on process again.

### 2.2.1. Log-on Process for Existing Users

If the name of an existing user is entered, Maximus prompts the user to enter the password for that user account. The user has up to five tries to enter the correct password. If none of the five attempts are successful, Maximus displays the `\max\misc\bad_pwd.mec` file. By default, this file allows the user to leave a message to the SysOp before the system hangs up.

Once Maximus validates the password entered by the user, it displays the `\max\misc\welcome.mec` file. This file can contain ANSI or *RIPscrip* graphics to be shown to the user.

### 2.2.2. Log-On Process for New Users

If the user enters the correct password, Maximus validates the user and displays the `\max\misc\welcome.mec` file.

If a non-existent username is entered, Maximus enters the “new user” state. In this state, Maximus first displays the `\max\misc\applic.mec` file, which normally gives the caller some information about the system. This file can also present an application form to be filled out by the user.

Maximus will then prompt for the user’s city and state/province, alias, phone number, and a number of other user settings.

Lastly, Maximus will display `\max\misc\newuser2.mec`. This file typically describes the system in more detail.

## 2.3. Command Stacking

Maximus allows the user to *stack* command responses by entering multiple words at an input prompt. This feature is normally used to expedite the log-on process, but command stacking can also be used for most other Maximus commands.

Without command stacking, a typical log-on sequence looks like this:

```
What is your name: John Doe
```

```
John Doe [Y,n]? y
```

```
Password: .....
```

Instead of entering each of these responses separately, all of the responses can be placed on the same line, as shown below:

What is your name: **John Doe;y;password**

Maximus also supports command line editing at most system prompts. After the user has logged on, and as long as ANSI or AVATAR graphics are enabled, the user can use the cursor keys to edit any of the text on the command line. Editing can be performed using the `<left>`, `<right>`, `<bs>`, `<del>`, `<ctrl-left>`, and `<ctrl-right>` keys.

To use the command line editing feature from remote, the user's terminal program must support either "Doorway mode" or VT-100 keyboard codes.

## 2.4. Guest Accounts

If the SysOp uses the user editor to create an account that has no password, Maximus treats it as a *guest account*. If a user logs on using a guest account, Maximus automatically skips the password prompt. In addition, Maximus prompts the guest user for a new set of configuration options at the beginning of every log-on, including editor preference, graphics support, and more.

This feature allows the SysOp to specifically create a single account for new users, even if the **Logon Level Preregistered** feature is enabled. This feature is useful when the SysOp wants potential users to fill out an application form (using the guest account) before granting access to the system.

## 2.5. The Main Menu

After displaying the log-on screens, Maximus also displays the text in the `\max\misc\bulletin.mec` file. This file typically contains system bulletins or other messages of interest to all users.

Following the system bulletins, the user is placed at the main menu. Although Maximus's menus are completely customizable, this section describes the commands that are found on the main menu in the default configuration.

### *Message Section*

This command takes the user to the *message section*. The message section is used to participate in public conference areas and to enter messages to other users. Please see section 2.6 for more information.

### *File Section*

This command takes the user to the *file section*. The file section contains libraries of files that can be downloaded (retrieved). The user can also upload (send)

## 8 2. Maximus Overview

files, search for files of a specific name, and display a file list. Please see section 2.7 for more information.

### *Change Setup*

This takes the user to the *change setup section*. This menu allows users to adjust settings in their user profiles. The user can change the screen width and length, select a default file transfer protocol, change telephone numbers, and more. Please see section 2.8 for more information.

### *Goodbye*

This option logs the user off and hangs up the phone. Even if the user does not log off using this command, Maximus will still update the user's user record. However, this command gives the user the opportunity to leave a comment to the SysOp.

Comments to the SysOp are placed in the message area specified by the **Comment Area** keyword in the system control file.

The subject used for the log-off comment can be set in the *comment\_fr* string in the Maximus language file. This string can include external program translation characters. Please see section 6 for more information.

### *Statistics*

This command displays the user's statistics, including the time of day, the length of the current call, the user's total time for the day, number of kilobytes uploaded and downloaded, and so on.

### *Yell*

This command allows the user to request a conversation with the SysOp. By default, this command plays a simple tune on the system speaker. (This tune can be configured in the `\max\tunes.bbs` file.) The local speaker can also be toggled on and off by pressing "!" at the local console while a user is logged on.

## **OS/2 only!**

Maximus can also play yell tunes on a SoundBlaster or SoundBlaster-compatible device. Please see section 18.11 for more information.

### *Userlist*

This command displays the list of all records in the user file. Users can exclude themselves from this list using the "InUserList" command on the Change Setup menu. In addition, the SysOp can modify the access control file to restrict the user list display to a certain set of privilege levels.



### *Version*

This command displays the Maximus version number and copyright information.

### *SysOp Menu*

This command takes the user to the SysOp menu. Normally, only the system operator or a trusted user will have access to this command. Please see section 2.9 for more information.

### *Chat Menu*

This command takes the user to the multinode chat menu. This menu allows the user to access all of Maximus's multinode features, including paging, toggling chat availability, and private/public chatting. (This command is used exclusively to allow users to communicate among themselves. The Yell command is used to chat with the SysOp.)

### *Who is On?*

This command displays a list of callers who are currently logged onto the system. The *Who is On?* display includes each user's name, status, and chat availability.

At the SysOp's discretion, other options and submenus can be added to the main menu. Please see section 18.8 for more information on adding menu options.

## 2.6. The Message Section

Maximus supports four distinct message area types: *local* areas, *NetMail* areas, *EchoMail* areas, and *conference* areas.

Local areas are used for messages that are local to your BBS. These messages can be public or private, but local messages do not get transmitted to other bulletin boards.

NetMail areas are used for private, point-to-point communication between users on two FidoNet-compatible systems. When entering a message in a NetMail area, Maximus prompts the user for additional addressing information, including the destination address of the message and (optionally) a number of message handling attributes. Maximus also maintains a "NetMail credit" account for each user that can be used when billing users for NetMail usage.

Conference areas and EchoMail areas (also known as *echoes*) are used for public conferences that are distributed among many systems. A message entered in an

EchoMail area is broadcast to all of the systems which carry that conference. To a Maximus user, an EchoMail area appears identical to a local message area, except that messages entered in EchoMail areas have network control information added to the end of the message. In addition, after a user enters a message in an EchoMail area, Maximus adds the name of that area to the **Log EchoMail** file. This file can be used later by a scanning program, such as Squish, to export that message to the rest of the network.

In addition to defining the message area type, a number of message area attributes can be assigned to each message area. A complete listing of these attributes can be found in section 18.6, but some of the more common attributes are given below:

#### *Pvt*

All messages entered in this area will be marked *private*. Private messages can only be read by the message sender and the addressee. (Users in a privilege level class that has the **ShowPvt** flag can also read private messages addressed to any user. Typically, these permissions are only granted to the SysOp.)

#### *Pub*

All messages entered in these areas are marked as public. Regardless of the addressee of the message, any user can read a public message. If both the **Pvt** and **Pub** attributes are specified, the user can enter either public or private messages.

#### *ReadOnly*

The SysOp can place messages in a read-only area, but normal users are unable to post messages in areas of this type. (Users in a privilege level class that has the **WriteRdOnly** flag can also post messages in this area.)

#### *Anon*

When a user enters a message in an area with the **Anon** attribute set, Maximus prompts the user to enter the **From:** field of the message. (In all other areas, Maximus automatically fills in the **From:** field with the user's real name.)

However, Maximus can optionally embed the user's real name within the message such that the SysOp can read it if necessary. This option is enabled by default. See the **Style NoNameKludge** flag in section 18.6 for more information.

#### *Attach*

Message areas with this attribute allow users to attach files to messages created in the area. Please see the description of the **Enter Message** command in sec-

tion 2.6 for more information. Also see section 5.2 for information on local file attaches.

The SysOp can also assign passwords to message and file areas, enable message tracking, add support for high-bit characters, and more. Access to individual message areas can also be granted based on a user's privilege level or name. Please see section 18.6 for more information.

### 2.6.1. The Message Menu

This section describes the commands that are found on the standard message menu:

#### *Area Change*

This command allows the user to select another message area. The user can select an area by name, but the “[” and “]” keys can be used to select the message areas which precede or follow the current area, respectively.

The “?” character displays a list of areas. For navigating within a hierarchical area structure, the “/” key moves to the top-level menu, and the “..” sequence ascends one level in the menu tree.

#### *Next*

This command displays the next message in the current area. After displaying a message with the **Next** command, pressing <enter> at the message area prompt automatically displays the next message.

#### *Previous*

This command displays the prior message in the current area. After displaying a message with the **Prior** command, pressing <enter> at the message area prompt will automatically display the prior message.

#### *Enter a Message*

This command allows a user to enter (post) a message in the current message area. Maximus first prompts the user to fill out the message header, including the *To* field, the *Subject* field, and the message attributes.

For users who have ANSI, AVATAR or RIPscrip graphics enabled, Maximus displays a message header with fields that can be filled out by the user. The <up> and <down> keys can be used to move between fields, as can the <shift-tab> and <tab> keys.

## 12 2. Maximus Overview

If the message area definition permits both public and private messages, Maximus allows the user to select either type of message. For users who have graphics enabled, the private flag can be toggled by pressing `<space>` when the cursor is on the attributes field (above the message date). The *P* key also has the same effect.

In areas which support file attachments, a file can be attached to a message by pressing the *A* key while the cursor is on the attributes field. (Maximus will prompt the user to upload the file after the message has been created.)

In Local areas, the user can obtain the system user list by pressing *?* at the **To** prompt. Maximus normally generates this list automatically from the system user file, but if a `\max\misc\userlist.mec` file exists, Maximus will display it instead of the real user list.

In NetMail areas, the attributes field can also be used to select other mail handling options. Entering a *?* at the attribute prompt gives a complete list of keys.

In addition, NetMail areas also allow the user to select the destination address for the message.

After entering the message header information, Maximus will run the system editor and allow the user to enter the body of the message.

### *Change a Message*

This command allows the user to modify an existing message. Although the SysOp can modify any message on the system, a normal user can only modify messages which:

- have not been read by the recipient,
- have not been sent (in the case of NetMail or EchoMail), and
- have the user's name in the **From** field.

Both the message header and the message body can be modified. When graphics are enabled, the user can also modify the message attributes.

### *Reply to Message*

This command allows the user to enter a response to the currently-selected message. The **Reply** command is similar to the **Enter** command; however, Maximus will automatically fill out the fields in the message header when doing a **Reply**. In addition, Maximus will also adjust the message reply chain so that the new message is marked as a reply to the original message.

Within the message editor, the user can also *quote* (copy) text from the original message into the newly-created reply.

#### *Read Non-Stop*

This command displays a number of messages all at once, with no pauses or prompts between messages. If the user last selected the **Next** command, Maximus displays all messages from the current message to the last message in the area. Otherwise, if the user last selected the **Prior** command, Maximus displays all messages from the current message to the first message in the area (in reverse order).

#### *Read Original*

This command examines the current message and finds the original message in the reply thread. (If the current message was a reply to another message, this command displays that other message.) Messages that are replies to other messages have a “\*\*\* This is a reply to #msg” line at the bottom of the message or a “- msg” field in the message header.

#### *Read Reply*

This command displays the message which is a reply to the current message, if any. Messages which have replies have a “\*\*\* See also #msg” line at the bottom of the message or a “+ msg” field in the message header.

#### *Read Current*

This command redisplay the current message.

#### *Browse*

This command scans any or all of the message areas on the system and retrieves selected messages. Browse acts as a powerful database engine for the message bases. Users can select a set of messages and operations to perform, and then have Maximus automatically identify and display the messages that were requested.

The first browse menu prompts the user to select a set of message areas. The user can select any of:

- ◆ the current area,
- ◆ all message areas,
- ◆ all tagged message areas,
- ◆ a wildcard specification (such as “comp.lang.\*”), or
- ◆ the current group (all areas which are on the same level in the message area hierarchy)

The second browse menu prompts the user to select a type of message. The user can specify:

- ◆ all messages,
- ◆ new messages (those which are above the user's lastread pointer),
- ◆ your messages (unread messages which are addressed to the user and which are above the user's lastread pointer), and
- ◆ *from* a certain message number (such as *from* message 500 to the end of the area).

Maximus also allows the user to perform a search based on keywords in the **To**, **From** and **Subject** fields, in addition to searching the message body. Complex searches can be defined by combining the logical **or** and **and** operators.

The third and final browse menu prompts the user to specify an operation to perform on the selected messages. Messages can be:

- ◆ listed (one to a line, with to/from/subject only),
- ◆ read (displayed in full, with the option to reply or kill each message), or
- ◆ packed (compressed into a QWK packet for download and off-line mail reading).

### *Tag*

This command allows users to select a subset of the message areas on the system. Each user can select his or her own set of personal message areas. This area selection is used to control the areas scanned by the **Browse** command and the off-line message reader.

### *Edit User*

This command reads in the current message, checks the **From** field, invokes the system user editor, and automatically positions the user editor on that user's user record. This option is useful for validating users, since a user's user record can be displayed at the press of a key. (This command is normally only available to SysOps.)

To invoke the user editor without seeking to a specific user, instead use the **User Edit** command from the SysOp menu.

### *Goodbye*

This option logs the user off and hangs up the phone. Even if the user does not log off using this command, Maximus will still update the user's user record. However, this command gives the user the opportunity to leave a comment to the SysOp.

Comments to the SysOp are placed in the message area specified by the **Comment Area** keyword in the system control file.

The subject used for the log-off comment can be set in the *comment\_fr* string in the Maximus language file. This string can include external program translation characters. Please see section 6 for more information.

#### *Main Menu*

This command returns the user to the main menu.

#### *Kill a Message*

This command allows the user to delete a message from the current area. A normal user can only delete messages which contain that user's name in either the **To** or the **From** field of the message to be deleted. However, the SysOp can delete any message on the system.

#### *Upload a Message*

This command allows the user to directly upload a text file as a message. This command is identical to the **Enter** command, except that Maximus prompts the user to start a file transfer protocol instead of invoking one of the system editors.

#### *Forward*

This command allows the user to copy a message in the current area and send it to someone else. Maximus prompts the user to enter the message number to be forwarded and the name of the addressee.

The user can also forward the message directly into another area by typing the area number when prompted. The **Forward** command also supports two special modifiers:

- Entering *FK* from the message area menu instructs Maximus to delete the original message after it has been forwarded.
- Entering *FB* from the message area menu instructs Maximus to send a "bombing run." Maximus prompts the user to specify a local filename that contains a list of message addressees. (In order to use this command, the user's privilege level must be equal to the level required to create a message from a file, as defined by the **Message Edit Ask FromFile** keyword in the system control file.)

The format of each line of the bombing run file is as follows:

**<username> <dest\_net/dest\_node> [-x]**

The **<dest\_net/dest\_node>** and **[-x]** fields are only used for NetMail messages and should be omitted for local bombing runs.

**-x** can be one of the switches shown below in Table 2.1:

**Table 2.1 Bombing Run Options**

| Switch | Description  |
|--------|--|
| -h     | The message should be marked as “hold for pickup.” |
| -c     | The message should be marked as “crash.”           |
| -n     | The message should sent normally (default)         |

In the username field, spaces in a user's name must be represented by underscores.

For example:

```
SysOp                1:225/337
Scott_Dudley         1:249/106   -c
Bob_Davis            1:106/114   -h
Vince_Perriello      1:141/191   -n
```

Title: AF  
Creator:

If you are performing a NetMail bombing run, it is courteous to send all messages directly to the destination (without routing your mail through other systems).

### *Hurl*

This command moves messages from one area to another. The **Hurl** command asks the user for the destination area and the number of the message to be moved.

### *Xport*

This command exports a message to a specific path and filename on the local system. (The **Xport** command is normally only available to SysOps.)

The exported message includes a copy of the message header. The message body is formatted for an 80-column screen.

To print a message on the printer, specify an Xport filename of “prn”.



In addition, other menu options can be placed on the message menu, including commands to call auxiliary menus, display text files, and run external programs. Please see section 18.8 for more information.

### 2.6.2. Message Editors

Maximus has two internal message editors: MaxEd, the full screen editor, and BORED, the line-oriented editor. Maximus also supports a single, SysOp-defined external message editor.

#### 2.6.2.1. MaxEd

MaxEd is the Maximus full screen editor. To use MaxEd, the user must have ANSI, AVATAR or RIP*scrip* graphics enabled, have a screen width of 80 columns, and have a screen length of at least 23 rows. The full screen editor has a number of advantages over the line editor, with the most obvious being that it is much easier to use. MaxEd is more like a word processor than a conventional BBS line editor; the cursor keys can be used to page through a message and insert or delete text at various points in the message.

For editing messages, MaxEd uses a mixture of WordStar and generic VT-100 keystrokes. A full list of the keys used by MaxEd can be obtained by pressing `<ctrl-n>` from within the editor.

When composing a reply to another message, text from the original message can also be *quoted* (copied) into the reply. The `<ctrl-k>R` sequence (or `<alt-q>` on the local keyboard) toggles the quote window display.

To scroll the quote window down by four lines, press either `<ctrl-c>` or `<pgdn>`.

To scroll the quote window up by four lines, press either `<ctrl-r>` or `<pgup>`.

To copy text from the quote window into the message, press either `<ctrl-k>C` or `<alt-c>`. Maximus will copy the text and automatically scroll the quote window down by four lines.

MaxEd also has its own menu that allows the user to modify the fields in the message header. This menu can be accessed by pressing either `<ctrl-k>H` or `<f10>`.

The options available on the MaxEd menu include:

*Continue*

This command returns the user to the MaxEd's message-editing screen.

*To*

This command allows the user to change the addressee of the message.

*Subject*

This command allows the user to change the subject of the message.

*From*

This command allows the user to change the **From** field of the message. The privilege level of this command should be set high enough so that only the SysOp can access this command.

*Handling*

This command allows the user to modify the message's attributes. Attributes such as the **Private** flag can be changed, in addition to NetMail attributes (such as **Crash**, **Hold** and **File Attach**). This option is normally only available to the SysOp.

*Read File*

This command allows the user to read in a file from the local hard drive and include it as part of the message. This option is normally only available to the SysOp.

**2.6.2.2. BORED**

BORED is the Maximus line editor. Unlike MaxEd, BORED does not require ANSI or AVATAR graphics, so it can be used by any user. (However, most users who have graphics capabilities will likely prefer to use MaxEd.)

BORED allows the user to enter a message one line at a time. If the user presses *<enter>* on a blank line, BORED displays the editor menu. The following editor commands are available:

*Save*

This command saves the message and returns to the message menu.

*Abort*

This command aborts (discards) the message and returns to the message menu.

### *List*

This command lists the message. Each line in the message is indicated by number.

### *Edit*

This command is used to replace text in a message. First, Maximus prompts the user to enter the line number which contains the text to be replaced. Next, the user enters the search text (the text which is to be replaced). Finally, the user enters the replacement text.

To insert text at the beginning of a line, simply press `<enter>` at the **Text to replace:** prompt.

### *Insert*

This command inserts a blank line in the message. The user can specify a line number to indicate where the blank line is to be placed.

### *Delete*

This command deletes a specified line in the message.

### *Continue*

This command allows the user to continue writing by appending new lines to the end of the message.

### *Quote*

For messages which are replies, this command allows the user to quote text from the original message. Maximus prompts the user to enter the starting and ending line numbers (in the original message) for the text to be quoted.

### *To*

This command allows the user to change the addressee of the message.

### *Subject*

This command allows the user to change the subject of the message.

*From*

This command allows the user to change the **From** field of the message. The privilege level of this command should be set high enough so that only the SysOp can access this command.

*Handling*

This command allows the user to modify the message's attributes. Attributes such as the **Private** flag can be changed, in addition to NetMail attributes (such as **Crash**, **Hold** and **File Attach**). This option is normally only available to the SysOp.

*Read File From Disk*

This command allows the user to read in a file from the local hard drive and include it as part of the message. This option is normally only available to the SysOp.

## 2.7. The File Menu

This section describes the commands that are found on the standard file menu:

*Area Change*

This command allows the user to select another file area. The user can select an area by name, but the “[” and “]” keys can be used to select the areas which precede or follow the current area, respectively.

The “?” character displays a list of file areas. For navigating within a hierarchical area structure, the “/” key can be used to go to the top-level menu, and the “..” sequence can be used to ascend one level in the menu tree.

*Locate*

This command allows the user to search all file areas on the system for a specific file.

Maximus first prompts the user to enter a text string. Next, it will go through all of the file areas on the system, and if it finds a match for that string (in either a filename or file description), it will display the name of the area and the corresponding file information.

The *L\** command instructs Maximus to search all file areas for new files. This command will display a list of files that have been added to the system since the last *L\** command was executed.

#### *File Titles*

This command displays a list of files in the current area. New files are identified by a flashing asterisk (\*) next to the file date.

If the user specifies an argument when invoking this command, the **File Titles** command displays only those files which contain that string in the filename or description fields. (However, the **File Titles** command only searches the current file area, whereas the **Locate** command searches all file areas.)

In addition, when the **More [Y,n,t,=]?** prompt is displayed during a file list, the “t” option (if present) allows the user to *tag* files for download. This option is only displayed if the user has an access level high enough to allow file downloading.

#### *View*

This command displays the contents of an ASCII file in the current area. Before displaying the file to the user, Maximus first checks the file to ensure that it contains only ASCII text.

#### *Goodbye*

This option logs the user off and hangs up the phone. Even if the user does not log off using this command, Maximus will still update the user’s user record. However, this command gives the user the opportunity to leave a comment to the SysOp.

Comments to the SysOp are placed in the message area specified by the **Comment Area** keyword in the system control file.

The subject used for the log-off comment can be set in the *comment\_fr* string in the Maximus language file. This string can include external program translation characters. Please see section 6 for more information.

#### *Main Menu*

This command returns the user to the main menu.

#### *Download*

This command allows the user to download one or more files from the system.

Maximus first prompts the user to select a file transfer protocol. (However, if the user has selected a default file transfer protocol from the **Change Setup** menu, Maximus will automatically skip to the next step.)

Maximus includes internal support for Xmodem, Xmodem-1k, Ymodem, Ymodem-G, SEALink, and Zmodem. Other transfer protocols can be added as external protocols.

After selecting a protocol, users are prompted to enter the names of the files to be downloaded, one file to a line. Files that were previously tagged using the **Tag** command are automatically included in the list.

When processing filenames entered by the user, Maximus automatically expands wildcards, including the “\*” and “?” characters.

In addition, if the FB utility is used to maintain a system file database, the filenames entered by the user can reside in any file area. (Otherwise, the user must change to the correct file area before selecting the **Download** command.)

In addition to entering filenames, the user can also:

- ◆ press <enter> to start the download,
- ◆ enter /q to abort the transfer without sending any files,
- ◆ enter /e to enter *edit mode*. This mode allows the user to list and delete files in the download queue, or
- ◆ enter /g to start the download and automatically hang up when the transfer is complete.

### *Tag*

This command allows the user to queue one or more files for later download. The **Tag** command is also accessible through the *t* command when performing a **File Titles** or **Locate** command.

To download files which have been tagged using this command, simply select the **Download** command and press <enter> at the **File(s) to download:** prompt.

### *Upload*

This command allows the user to upload (send) files to the system. If no default file transfer protocol is defined, Maximus prompts the user for the protocol to be used for the upload. If the user selects Xmodem or Xmodem-1K, Maximus also prompts the user for the name of the file to be uploaded. (With all of the other file transfer protocols, the filename is automatically transmitted by the sending terminal program.)

Maximus will then start the upload. After the transfer is complete, Maximus will prompt the user to enter a description for each file that is uploaded.

Maximus can use the upload filename to automatically screen out certain types of uploads. The `\max\badfiles.bbs` file contains a list of files to be ignored. This list of files can include wildcards. A sample `badfiles.bbs` could look like this:

```
MAKE$$$ .TXT
MAKECASH. *
* .RBS
* .GBS
* .BBS
* .GIF
* .JPG
* .TIF
```

#### *Statistics*

This command displays the user's statistics, including the amount of time that the user has been online for the current call, the time online for the day, number of kilobytes uploaded, number of kilobytes downloaded, and so on.

#### *Contents*

This command displays the internal file catalog of a compressed file. The **Contents** command can display the directory of any `.zip`, `.arc`, `.pak`, `.arj` or `.lzh` file

#### *Raw Directory*

This command displays a listing of all files in the current area, including files which are not listed in the `files.bbs` catalog for that area. This command is normally only available to the SysOp.

#### *Override Path*

This command allows the user to override the download path for the current file area. For example, the download path can be overridden to `c:\max` to allow a privileged user to download files from the Maximus system directory. This command is normally only available to the SysOp.

#### *Hurl*

This command moves a file from one area to another. Maximus prompts the user for the name of the destination area and the name of the file to be moved. This command is normally only available to the SysOp.

*Kill File*

This command deletes a file from the current file area. Maximus will prompt the user for the name of the file to be deleted. Maximus will ask the user to confirm the name of the file to be deleted; if the user answers “n” to this prompt, Maximus will give the user the option of leaving the physical file alone and removing the entry from the file listing. This command is normally only available to the SysOp.

## 2.8. The Change Setup Menu

This menu allows the user to modify settings in the user profile. The following commands are available on the standard **Change Setup** menu:

*City*

This command modifies the user’s “City” setting.

*Phone Number*

This command modifies the user’s “Phone Number” setting.

*Alias*

This command is designed for use on systems that support aliases. This command allows a user to change his/her alias.

*Password*

This command changes the user’s password. The user is first prompted to enter the old password. The user is given five chances to correctly enter the old password before Maximus hangs up.

Next, the user is prompted to enter the new password. The user must enter the new password twice to prevent accidental typing errors.

*Help Level*

This command selects the system help level. Maximus supports the help levels shown below in Table 2.2:

**Table 2.2 System Help Levels**

| Type    | Description       |
|---------|-------------------|
| NOVICE  | Full menus        |
| REGULAR | Abbreviated menus |



---

EXPERT      No menus

---

### *Nulls*

This command selects the number of NUL characters which are transmitted after every line of text. Most users will not need to use this command.

### *Width*

This command changes the assumed screen width. This value is used by Maximus when displaying system menus and when drawing the full-screen editor display.

However, for local users, Maximus will automatically detect the local screen size and adjust the “current width” value accordingly. The local screen size value overrides (but does not update) the value in the user file.

### *Length*

This command changes the assumed screen length.

### *Tabs*

This command toggles the translation of tab characters. Maximus normally sends tab characters directly to the user’s terminal. However, if the user’s terminal program does not support tab characters, this option can be disabled.

### *More*

This command toggles the “More [Y,n,=]?” prompts on and off.

### *Video Mode*

This command selects the user’s video mode. Maximus supports the following video modes:

- ◆ TTY
- ◆ ANSI
- ◆ AVATAR

RIPscrip graphics mode can be toggled by a separate menu option.

### *Full-Screen Editor*

This command toggles the use of the MaxEd full-screen editor. If MaxEd is turned off, BORED is used for message editing.

*IBM Graphics*

This command toggles the use of IBM “extended ASCII” characters. DOS and OS/2 based computers support an “extended” 8-bit character set, including characters for box-drawing and block graphics. Most non-IBM systems do not support these extended ASCII characters, so Maximus can be configured to map extended ASCII characters into normal ASCII characters.

*Hotkeys*

This command toggles the hotkeys setting. Turning on hotkeys instructs Maximus to process keystrokes as soon as they are received (without requiring an *<enter>* after most commands).

*Language*

This command selects an alternate system language. Maximus supports up to eight different language files, and the user can switch between installed language files at any time.

*Protocol*

This command selects a default file transfer protocol. When a default protocol is selected, the **Download** command immediately displays the **File(s) to download:** prompt instead of asking the user to select a protocol.

*Archiver*

This command selects a default archiving program. This allows the user to select a commonly-used archiver for compressing QWK mail packets.

*Show in Userlist*

This command toggle’s the displaying of the user’s name in the system user list. Users are displayed in the user list by default, but this command can be used to hide the display of the user’s name and last-call information.

*Full-Screen Reader*

This command toggles the use of the full-screen reader (FSR). When the FSR is enabled, Maximus will display a stylized blue box at the top of the screen when reading messages. This header includes information from the **To**, **From** and **Subject** fields, information on the message reply links, and net/node information.

*RIP*

This command toggles the use of *RIPscrip* graphics. When this command is selected, Maximus will test the remote user's terminal program to ensure that it supports *RIPscrip*. If *RIPscrip* graphics support is not detected, Maximus will not enable *RIPscrip* graphics.

*Quit*

This command returns to the main menu.

## 2.9. The SysOp Menu

This menu contains commands that are normally only accessible to the SysOp.

*User Editor*

This invokes the Maximus internal user editor. This command is normally only available to the SysOp. Please see Appendix E for more information.

*OS Shell*

This command invokes either a local or a remote shell to the operating system. Note that *<alt-j>* can always be used from the local console to shell to the operating system.

**OS/2 only!**

When running external programs that are not specifically designed to run as doors, OS/2 users should use the MaxPipe program to invoke the command, rather than redirecting input with the "<" and ">" characters.

**DOS only!**

Under DOS, if you are using an alternate command processor (such as 4DOS or NDOS), the entry for this command (in the menus control file) must be changed to reflect the name of your command processor.

## 2.10. The Chat Menu

Maximus supports an internal multinode chat facility. Users on different nodes can hold private discussions with one another, and users can even engage in large group discussions on a "virtual CB channels."

*CB Chat*

The **CB Chat** function allows two or more users to participate in a real-time multinode chat. Messages can be sent back and forth to all users who are par-

icipating on a specific CB channel. Messages are sent to other users one line at a time. Maximus supports up to 255 virtual CB channels.

#### *Page User*

The **Page User** command allows a user to initiate a private chat with another node. Selecting **Page** instructs Maximus to send a “chat request” message to the specified node number. Maximus will then place the paging user in chat mode and wait for the other user to respond to the page.

#### *Answer Page*

The **Answer Page** command is used in conjunction with the **Page User** command. After a user receives a page request from another node, the paged user can select **Answer Page** to engage in a private chat with the first user.

#### *Toggle Status*

The **Toggle Status** command allows a user to toggle the chat availability flag. Users who are unavailable for chat cannot be paged using the **Page User** command.

## 2.11. The Off-Line Reader Menu

The Off-Line Reader menu is the key to Maximus's internal QWK mail packer. Commands on this menu can be used to select a default protocol and archiving program, select a set of message areas, download messages from that set of areas, and upload reply packets.

#### *Tag Area*

The **Tag Area** command (equivalent to the command of the same name on the message menu) allows the user to select a set of message areas to be processed by the **Browse** and **Download** commands.

#### *Download New Messages*

The **Download** command packs all new messages in the user's set of tagged areas. The messages are then compressed into an archive and sent to the user.

#### *Upload Replies*

The **Upload Replies** command allows the user to upload a **.rep** reply packet. Max automatically determines the program used to compress the **.rep** file, de-

compresses the reply packet, and places the uploaded messages into the appropriate areas.

### *Protocol Default*

The **Protocol Default** command allows the user to select a default file transfer protocol.

### *Archiver Default*

The **Archiver Default** command allows the user to select a default archiving program.



---

## 3. Installation

This section describes how to install a new copy of Maximus 3.0. (To upgrade from Maximus 2.0, simply run the install program and follow the directions.)

### 3.1. Step 1: Unpacking the Distribution Files

If you purchased a copy of Maximus, all of the required files are on the distribution disk and nothing more needs to be done. Skip to step 2 for information on running the install program.

Otherwise, if you obtained Maximus electronically, the system consists of three separate archives, as shown below in Table 3.1:

**Table 3.1 Maximus Distribution Files**

| File        | Description   |
|-------------|---|
| max300r.zip | DOS (real-mode) executables.                        |
| max300p.zip | OS/2 (protected-mode) executables.                  |
| max300c.zip | Common executables and files for both DOS and OS/2. |

To install Maximus, you need **max300c.zip**, plus either or both of **max300r.zip** or **max300p.zip**, depending on the operating systems that you wish to use.

The install program will examine the files that are available, and if both the DOS and OS/2 versions of the executables are present, it will allow you to select either or both versions for installation. Of course, you can also install Maximus for only one of the supported operating systems.

The first step in the installation is to unarchive all of the required **.zip** files into a temporary subdirectory. (The install program will move the files from the temporary directory to a permanent directory of your choice, so this temporary directory can be located anywhere on your system.)

To decompress the **.zip** files, you need either the commercial “PKUNZIP” program from PKWare or the freeware “unzip” program from InfoZip.

The following files are contained inside the **.zip** archives:

- an ASCII version of the system documentation,
- the install program,

- the program license agreement, and
- a number of files with a **.fiz** extension. Most of the programs in the distribution kit are packed using the proprietary FIZ compression algorithm, and the only way to extract these files is to run the install program.

### 3.2. Step 2: Running the Installation Program

To execute the install program, run **install.exe** from the install disk (or from the temporary directory, for the electronic distribution version).

After displaying some copyright information, the install program will prompt you for the type of install to perform. Select the **New Install** button here, since these installation instructions only describe new installations.

In the next dialog box, the install program will prompt you for a number of system paths. Aside from changing the drive letter or root name of the **\max** directory hierarchy, inexperienced users should leave these paths alone.

Next, the install program will prompt you for some basic information about your system, including the BBS name, the name of the SysOp, and information about your communications hardware.

In the dialog box, simply type text in the appropriate boxes, and use **<tab>** (or click with the mouse) to move between fields. To select a particular option from a radio button group, use the **<left>** and **<right>** keys to cursor over to the appropriate location and press **<space>** to select that option. Press **<enter>** or click on the OK button when you have specified the correct values.

#### OS/2 only!

Most of the executables in the OS/2 version of Maximus end with the letter “p.” This “p” signifies that the executable runs in *protected mode* and is a native OS/2 application. Adding an extra “p” to the filename allows both Maximus-DOS and Maximus-OS/2 to be installed into the same directory.

However, to keep this manual concise, only the base name of an executable is mentioned, without the trailing “p.” When working through the installation, keep in mind that when the documentation refers to an executable filename, a “p” should be added for OS/2 users.

For example, while the DOS version of the “ANS2BBS” utility is called **ans2bbs.exe**, the OS/2 version is called **ans2bbbsp.exe**.



### 3.3. Step 3: Configuring your Modem

This section describes how to configure your modem to work with Maximus and other related software packages.

The modem is your BBS's gateway to the rest of the world, but it can also be one of the most difficult components to configure correctly. Due to the wide variety of modem manufacturers and products, this manual cannot possibly cover all aspects of installing and configuring modems. However, this documentation describes a common set of modem commands that are supported by most popular modems.

To run smoothly, Maximus requires a Hayes-compatible modem. Although it may be possible to use Maximus with a non Hayes-compatible modem, only Hayes-compatible modems are officially supported.

When setting up your modem, the first step is to determine how your modem handles the Data Carrier Detect (DCD) signal. DCD is a signal sent by the modem to your computer to indicate when a connection has been established.

When your modem is idle, DCD is normally in the *low* state. However, as soon as a user connects to your modem, DCD becomes *high*. Maximus uses the DCD signal to determine when a user connects with the system, and it also uses DCD to determine when a caller hangs up.

Unfortunately, the default settings of many lower-speed modems instruct the modem to always set the DCD signal high, regardless of whether or not the connection is active.

To ensure that your modem is reporting the DCD signal properly, you must check the configuration of your modem. Depending on your modem type, this can be done in a number of ways:

*1200 bps modems.* If your modem is 1200 bps or slower, chances are that your modem's DCD handling is controlled by DIP switches. DIP switches are the small plastic toggles on the front, rear or bottom of your modem. (One or more panels may need to be removed to access these switches.)

On most 1200 bps modems, DIP switch #6 controls the DCD signal. For proper operation, this switch should be in the *up* position so that the modem reports the true DCD value. (However, some modems are different, so please consult your modem documentation before changing any DIP switches.)

In addition, you may also need to change one of the other DIP switches so that your modem sends back verbal result codes (as opposed to numbers). The DIP switch to control these result codes is manufacturer-dependent, so you will again need to consult your modem's manual to determine which switch to check.

2400 bps, 9600 bps, 14.4 kbps, 19.2 kbps, or 28.8 kbps. If your modem is 2400 bps or faster, the configuration process can usually be performed using a standard terminal program (including Procomm Plus, Telix, Crosstalk, and others).

After loading your terminal program and configuring it for the correct communications port, type in the command “AT” and press <enter>. If everything is well, your modem should return an “OK” response.

After you have established that your modem is working correctly, enter the command “AT&C1&S1&D2&W” and press <enter>. (If this command does not work, try omitting either or both of the “&C1” or the “&S1” strings, since some modems do not support these settings.) This command configures your modem so that DCD always reflects the modem's actual state, and it also configures your modem's DTR handling so that Maximus can use the DTR signal to end a session and hang up on a user.

*External modems.* If you have an external modem, one other potential problem is the modem cable. If your cable does not have the correct signals wired through, your modem may still behave as if DCD is set incorrectly, regardless of DIP switch settings.

The best way to determine whether or not your modem cable is working correctly is simply to borrow a cable from another SysOp with a working BBS and try it on your system.

If you determine that your cable is at fault, most computer stores or service centers can order or manufacture a modem cable. The wiring specifications for modem cables are:

*DB25 connectors (25 pins).* This is the most common type of modem cable. As a minimum, pins 2 through 8 and pin 20 must be wired straight through.

*DB9 connectors (9 pins).* All nine pins must be wired straight through.

## 3.4. Step 4: Installing Communications Drivers

### 3.4.1. OS/2 Communications Drivers

#### OS/2 only!

The following paragraphs are for OS/2 users only. DOS users can skip to section 3.4.2.

Unlike DOS, OS/2 does not require a FOSSIL driver. OS/2 includes its own device driver to handle serial I/O. Unfortunately, the device driver included with OS/2 does not work correctly in all situations. (Under OS/2 2.x, the supplied COM.SYS driver sometimes uninstalls itself after certain communication errors occur.)

Fortunately, a number of third-party device drivers are available:

The 16-bit COM16550.SYS device driver can be used instead of the standard IBM driver. An evaluation version of COM16550 is bundled with Maximus, but COM16550 is a third-party product that is not sold or supported by Lanius. Please see the documentation in the **\max\com16550.zip** file for more information.

In addition, a third-party, 32-bit device driver called SIO.SYS has many more features than COM16550 and can operate at much higher speeds. SIO.SYS is not provided with Maximus, but it can be obtained from the Hobbes OS/2 archive at *ftp.cdrom.com*. SIO can also be found on most bulletin board systems that offer OS/2 support.

Please note that COM16550 has a maximum speed of 19.2 kbps, whereas SIO.SYS has a maximum speed of 57.6 kbps. For this reason, if you are running a V.34 or V.FC modem, SIO.SYS is preferable to COM16550.

In addition, Maximus runs with any serial device driver that supports the standard OS/2 “ASYNC IOCtl” interface. This means that Maximus-OS/2 can be used with intelligent serial cards such as the DigiBoard or the ARCTIC card.

#### 3.4.2. DOS FOSSIL Drivers

Under DOS, Maximus requires an external serial port driver called a *FOSSIL*. FOSSIL is an acronym which stands for “Fido/Opus/SEAdog Standard Interface Layer.” The FOSSIL handles all of Maximus's low-level serial communication needs, including sending and receiving characters.

Using a FOSSIL allows Maximus to be used on a wide range of serial port hardware without modifying the Maximus core. (For example, third-party vendors have developed FOSSIL drivers that support the DigiBoard intelligent serial card.)

Maximus ships with a copy of Unique Computing Pty's BNU FOSSIL driver. BNU supports most non-intelligent serial ports. If BNU is not suitable or will not run on your system, other FOSSIL drivers are available. Some of the most common FOSSILs are X00 and OpusComm.

There are two separate types of FOSSIL drivers. Some FOSSIL drivers, such as OpusComm and BNU, are loaded as Terminate and Stay Resident (TSR) programs in the **c:\autoexec.bat** file. Other drivers, including X00, are loaded in the **c:\config.sys** file. Although different FOSSIL drivers have different set-up instructions, it is fairly easy to install a FOSSIL for a basic configuration.

*OpusComm installation.* To install OpusComm on COM1, simply insert the following commands at the beginning of your AUTOEXEC.BAT:

```
opuscomm
ocom_cfg L1=19200           (see note below!)
```

Ensure that **opuscomm.com** and **ocom\_cfg.exe** are on your current PATH, or else these commands will not work.

The second “ocom\_cfg” line locks port 1 at a speed of 19200 bps. This line is required for 9600+ bps modems. (Note that OpusComm does not support speeds faster than 19.2kbps.)

This line is only required if you are using a 9600+ bps modem. Users with 2400 bps or slower modems must not include the call to **ocom\_cfg**.

*BNU installation.* To install BNUcom on COM1, simply insert the following command at the beginning of your AUTOEXEC.BAT:

```
bnu -l0=38400
```

Ensure that **bnu.com** is on your current PATH, or else this command will not work.

The “-l0=38400” command instructs BNU to lock port 0 (COM1) at a speed of 38.4 kbps. This parameter is required for 9600+ bps modems. Users with 2400 bps modems must omit the “-l0=38400” parameter.



BNU uses 0-based COM port numbers. To lock the speed of COM1, use “-l0=*speed*”; to lock the speed of COM2, use “-l1=*speed*”; and so on.

*X00 installation.* To install the X00 driver on COM1, simply insert the following command at the beginning of your CONFIG.SYS:

```
DEVICE=X00.SYS B,0,19200
```

Ensure that X00.SYS is in your **c:\** directory, or else this command will not work.

The “B,0,19200” parameter instructs X00 to lock COM1 at a speed of 19200 bps. This parameter is required for 9600+ bps modems. Users with 2400 bps modems must omit the “B,0,19200” parameter.



X00 uses 0-based COM port numbers. To lock the speed of COM1, use “B,0,*speed*”; to lock the speed of COM2, use “B,1,*speed*”; and so on.

### 3.4.3. Checklist for high-speed modems

When using a high speed modem (9600 bps or above, including any modems which support V.32, V.32bis, V.32ter, V.FC, or V.34), the modem normally communicates with the host BBS at a fixed speed, regardless of the speed of the user’s mo-

dem. For this reason, if you are running a high-speed modem, you must instruct Maximus to talk to the modem at a fixed speed.

To use a fixed port speed, you must:

- ensure that the “&B1” parameter is included in your modem initialization string,
- ensure that CTS flow control is enabled (using the “Mask Handshaking CTS” option in the system control files), and
- use the *-speed* command line parameter when starting Maximus. (This parameter is only required when running Maximus in a mode that handles remote callers. This parameter is not required when starting Maximus in local mode.)

For example, to instruct Maximus to wait for a caller and use a locked port rate of 38.4 kbps:

```
max -s38400 -w
```

The “-s38400” parameter instructs Maximus to talk to the serial port at 38.4 kbps only. The modem itself will handle all rate negotiation with the remote system.

### 3.5. Step 5: Editing Configuration Files

In order to run Maximus on your system, you need to make several changes to your system configuration files, including **config.sys** and **autoexec.bat**.

For example, Maximus and related utilities always need to know how to find the main Maximus system directory. To accomplish this, these utilities examine the system environment for the **MAXIMUS** environment variable. This variable must always point to the master system configuration file.

**OS/2 only!** DOS users should skip the next few paragraphs.

To set up the Maximus environment variable under OS/2, you must add the following line to the end of the **config.sys** file on your OS/2 boot drive:

```
SET MAXIMUS=C:\MAX\MAX.PRM
```

This example assumes that Maximus was installed in the **c:\max** directory. If you installed Maximus somewhere else, substitute the appropriate path for “c:\max”.

After making this change to **config.sys**, you must reboot the computer for the change to take effect.

**DOS only!** OS/2 users should skip to the next section.

To configure Maximus to run under DOS, you must add a number of lines to your **c:\config.sys** file. An ASCII editor, such as the standard DOS **edit.com**, can be used to edit this file.

The first line to be added to **config.sys** is the “buffers=” statement. If there is no **BUFFERS** line in your **config.sys** file, simply add the following line to the end of the file:

```
BUFFERS=30
```

However, if a **BUFFERS** line already exists, ensure that it specifies a value of at least 30.

Next, the “files=” statement must be added. If there is no **FILES** line in your **config.sys** file, simply add the following line to the end of the file:

```
FILES=40
```

However, if a **FILES** line already exists, ensure that it specifies a value of at least 40.

Finally, if you plan to use Maximus in a multinode environment, Maximus also requires the **share.exe** program. (Even in single-node environments, **share.exe** is still strongly recommended.)

To install **share.exe**, simply add the following line to the end of **c:\autoexec.bat**:

```
share
```

*Note for Novell users.* Installing **share.exe** is not completely necessary. As long as you load **int2f.com** after running the Netware shell, you do not need to load **share.exe**.

*Note for Windows for Workgroups and Windows 95 users.* Windows comes with a SHARE-compatible VxD, and as long as you run Maximus exclusively within the Windows for Workgroups or Windows 95 environments, you do not need to install **share.exe**.

Finally, to set up the system environment variable under DOS, you must add the following line to the end of the **c:\autoexec.bat** file:

```
SET MAXIMUS=C:\MAX\MAX.PRM
```

This example assumes that Maximus was installed in the **c:\max** directory. If you installed Maximus somewhere else, substitute the appropriate path for “c:\max”.

Once you have made all of these changes and saved the configuration files, you should press `<ctrl-alt-del>` to reboot the computer. Unless you reboot now, changes made to **config.sys** and **autoexec.bat** will not take effect.

### 3.6. Step 6: About Control Files

This is the most complicated step in setting up a Maximus system. Four text-based configuration files control most of the operations of your system: **max.ctl**, **msgarea.ctl**, **filearea.ctl** and **menus.ctl**.

**max.ctl** controls almost everything about your system, from the name of the log-on display file to the “time reward” given to users who upload files.

**msgarea.ctl** controls all of the message areas that are accessible to users.

**filearea.ctl** controls all of the file areas that are accessible to users.

**menus.ctl** controls all of the menus and options that can be selected by users.

All of these text files are stored in an ASCII format, so the DOS **edit.com** or the OS/2 **e.exe** can be used to modify these files.

These files contain a number of fairly complicated commands, but these control files give you control over even the most minute aspects of your BBS. However, the power to tailor your BBS to a very specific set of needs also makes it relatively easy to configure your system incorrectly. Consequently, new SysOps are advised to refrain from making too many modifications to the control files until the basic BBS is up and running.

The installation program will have already customized most of the system control files, so no specific modifications are required at this point.

### 3.7. Step 7: Compiling the Control Files

Every time you modify **max.ctl** or any of the other control files, you must run SILT, the Maximus control file compiler. If you make changes to your control files and forget to run SILT, Maximus will not recognize the changes that you have made.

Title: Af  
Creator:

The system installation program will have already compiled your control file for the first time. However, it is a good idea to compile it again here, just so that you can learn how to run SILT..

Compiling your control files with SILT is easy; just enter the following from the command prompt:

```
silt ctlname
```

where **ctlname** is the name of your main system control file. In the default installation, the main control file is always called **max**, so most users only need to type “silt max”.

When SILT runs, it automatically compiles all of the control files, including **menus.ctl**, **filearea.ctl**, **msgarea.ctl**, and a number of other control files. These secondary control files cannot be compiled alone; you must always run SILT on the main control file, regardless of which secondary control file was changed.

When SILT runs, it will scan through all of the control files and write a compiled version of the system information to **max.prm**, **marea.dat**, **farea.dat**, and a number of other system data files.

If you made mistakes in the control files, such as misspelling a keyword or omitting a parameter, SILT will warn you about these mistakes. To correct these problems, use either the DOS **edit.com** or the OS/2 **e.exe** to load the control file, move to the problem line number, fix the error, and then recompile using SILT.

### 3.8. Step 8: Starting Maximus

Once you have compiled your control files, you are finally ready to start Maximus. Although your BBS is still fairly generic, it is now usable.

To start up Maximus for the first time, enter the following sequence of commands. This example assumes that you have installed Maximus in the **c:\max** directory:

```
c:
cd \max
max -c
```

The “-c” parameter tells Maximus to create a new user file, so you should only do this the first time you run Maximus. However, if you are converting from another BBS program, you should run the CVTUSR program before entering the above command. (See section 8.3 for more information on the CVTUSR program.)



By default, Maximus encrypts all passwords in the Maximus user file. While this adds an extra layer of security to your system, it means that you will be unable to convert your Maximus user file to the file format of any other BBS program. If you want to disable the password encryption, see the **No Password Encryption** feature in the system control file.)

After entering “max -c”, Maximus will display a copyright banner and print out a warning about the lack of an existing user file. Maximus will then display the prompt: “Your Name Here [Y,n]?”, where “Your Name Here” is the name entered



in the “SysOp” box in the installation program. Type the letter “Y” to continue your logon.

After doing this, Maximus will display some text that describes your BBS. This text is contained in the `\max\misc\applic.mec` file. (Files with an extension of `.mec` will be discussed in greater detail later in this document.)

Maximus will also prompt you for a few pieces of information, including your city, phone number, and password. Maximus will also prompt you for ANSI screen control support, the MaxEd editor, IBM-PC characters, hotkeys, and *RIPscrip* graphics support. Answer “y” to the first four prompts, but answer “n” to the *RIPscrip* graphics question.

After Maximus has finished configuring your account, it will display a welcome screen and a bulletin file. Finally, it will place you at the main system menu. All of the screens that you see are completely customizable. The steps required to customize these files are described later in this manual.

Now that Maximus is working, you will probably want to look around for a while. Feel free to explore the different features of your new BBS. If you want to send some test NetMail messages, try going into the user editor (from the SysOp menu) and giving yourself some NetMail credit.

When you have finished looking around at your new BBS, type “g” from any menu to log off, and Maximus will exit back to the command prompt.

### 3.9. Step 9: Support for Remote Callers

To handle non-local callers, Maximus must be run from a batch file. Unlike local log-ins, Maximus requires a batch file to recycle after remote callers.

There are two mutually-exclusive methods of running Maximus:

- Max can be run using the internal Waiting For Caller (WFC) subsystem. WFC takes care of answering the phone and talking to the modem, so no external programs are required.
- Maximus can also be run under a *front end*. A front end takes care of answering the phone and performing additional processing. Front ends are normally only required for FidoNet or UUCP support. If you are running a standalone system, you do not require a front end.

In some cases, systems that run multiple nodes may wish to run a front end only on a subset of the BBS lines. In many cases, only one node needs to run a front end program, while the others can all use the internal WFC module. The command line

is used to configure the system for WFC or front end operation, so no changes to the control files are required to support this.

### 3.9.1. Running the *MODE* command

#### OS/2 only!

If you are running Maximus under OS/2, special care must be taken to set up the communications port before calling Maximus. In most cases, the OS/2 **MODE** command is used to configure the port. **MODE** is used to set the port speed, flow control characteristics, and buffer settings.

In most cases, the following command should be issued before trying to run Maximus with a remote caller. This command must be entered all on one line without spaces:

```
MODE COMport: speed, n, 8, 1, , TO=OFF, XON=ON,
      IDSR=OFF, ODSR=OFF, OCTS=ON, DTR=ON, RTS=HS
```

In the above command, *port* is the 1-based com port number of your Maximus system. *speed* is the maximum communications rate supported by your modem. This line should be included in the command file that starts Maximus.

For example, to configure com port 1 for 38.4 kbps, the following command should be used:

```
MODE COM1: 38400, n, 8, 1, , TO=OFF, XON=ON, IDSR=OFF,
      ODSR=OFF, OCTS=ON, DTR=ON, RTS=HS
```

If your modem has special flow control requirements, please see the documentation for the **MODE** command for more information. (On-line help is available by typing "help mode" from the OS/2 command prompt.)

### 3.9.2. Using *WFC* Mode

When run in this mode, Maximus handles incoming callers on its own. The first step in enabling WFC mode is to use the "-w" command line parameter. To start Maximus in the most basic WFC mode, the following command is used:

```
max -w
```

"-w" instructs Maximus to enter WFC mode. After this command is executed, Maximus will load up, display a few windows, initialize the modem, and then wait for an incoming call. Maximus will automatically detect the incoming caller's speed and switch speeds automatically.

By default, Maximus is set up to run on any Hayes-compatible modem. If the WFC subsystem does not seem to be answering the phone correctly, read through the

comments in the **Equipment** section of the system control file. Some of the modem command strings may need to be modified, but almost all modems can be made to work with Maximus.

In addition to the standard “-w” switch, you can also use the “-b” (baud rate) and “-p” (COM port) switches to specify an alternate port number and maximum baud rate for the current node, overriding the defaults in the control file.

For example, to start WFC on port 2 with a baud rate of 38400, specify the following command:

```
max -w -p2 -b38400
```

To run WFC mode with a high-speed modem (9600+ bps), you must use a locked COM port. The “-s” command line parameter tells Maximus the speed to use for locking the port.

For example, the following command is used to run Maximus with a high-speed modem on COM2 (locked at 38.4 kbps):

```
max -w -p2 -s38400
```

No matter which options you use, the command line must always include a “-w” if you wish to handle remote callers. This command must be placed in the batch or command file that starts Maximus. (The batch/command file is described in more detail later in this section.)

The WFC module can also handle timed events which cause Maximus to wake up on specific days of the week or at a specific time of day. Please see section 5.1 for more information on WFC mode.

### 3.9.3. Using an External Front End

In this mode, Maximus will not answer the telephone by itself. You must obtain a third-party “front end” or “mailer” program to handle incoming calls. There are at least a dozen freeware/shareware FidoNet front ends for DOS, and two or three similar programs for OS/2. (At the time of this writing, the two most common front ends were BinkleyTerm and FrontDoor.)

Although setting up your front-end mailer is beyond the scope of this document, you will find several sample batch files for different front end mailers in Appendix G.

Your mailer's documentation may give some specific instructions for interfacing it with a Maximus system; if so, you should just follow those directions. If not, read on.

When Maximus starts up with a caller already on-line, it expects to be given a minimum of one parameter: “-*bspeed*”, where *speed* is the speed of the caller.

Generally, these parameters are passed from the mailer via the **spawnbbs.bat** or **exebbs.bat** files.

### OS/2 only!

Under OS/2, Maximus also expects to be passed the COM port handle from the calling program. This means that the minimum requirements for starting Maximus are:

```
maxp -bspeed -phandle
```

where *speed* is the speed of the caller and *handle* is the OS/2 file handle for the communications port. Unlike under DOS, the -p parameter is a COM port handle (which is not the same value as the COM port number). This means that this value must be passed to Maximus by your front end mailer.

#### 3.9.4. Maximus Errorlevels

Although the preceding commands allow Maximus to handle one remote caller, either through the WFC subsystem or through a front end program, Maximus normally exits back to the command prompt after it processes a caller.

After Maximus terminates, it needs to tell your system what to do next. For example, if a user entered a message in an EchoMail area, you may want to use an external utility (such as Squish) to export that message, or you may wish to run some sort of logging utility.

To provide for these external programs, Maximus tells the operating system to set a numeric value called an *errorlevel*. As mentioned earlier, Maximus supports several different errorlevels for various types of events, including users entering EchoMail messages, users entering NetMail messages, logging off before the user enters a name, and so on.

In several places throughout the control files, you can instruct Maximus to use a certain errorlevel when a given event occurs. Errorlevels are always numeric, and they always have a value from 1 to 255. In most cases, the errorlevels values do not need to be modified, but you can change them if you must. (However, Maximus reserves errorlevels 1 through 4 to indicate errors, so you should not use these values in the control file.)

Once Maximus is set up to use errorlevels, you must also write a batch file to detect the errorlevel returned by Maximus and take the appropriate action.

The following statement is used to test an errorlevel in a **.bat** file (DOS) or in a **.cmd** file (OS/2):

```
if errorlevel erl action
```

*erl* is a number which corresponds to the errorlevel value for the event, as specified in the system control file.

*action* is an action that is to be performed when the specified errorlevel is detected. This *action* can consist of any normal batch file statement.

However, if you wish to test for multiple errorlevels, be warned that both DOS and OS/2 examine errorlevels using a greater-than-or-equal-to comparison. This means that the following statement:

```
if errorlevel 10 echo Hi!
```

will be executed if Maximus sets an errorlevel value of 10 or greater.

For this reason, if you have more than one errorlevel to process, the group of errorlevel statements must be listed in *descending* order. For example, to check for errorlevels 1, 3, 9, 10, 11 and 12, your batch file would look like this:

```
max -w -p1 -b38400

if errorlevel 12 echo Do operation "A" here.
if errorlevel 11 echo Do operation "B" here.
if errorlevel 10 echo Do operation "C" here.
if errorlevel 9 echo Do operation "D" here.
if errorlevel 3 echo Do operation "E" here.
if errorlevel 1 echo Do operation "F" here.
```

Also, remember that *all* programs modify the errorlevel value when they are run. In the example given above, if you wanted to run a program called **abcd.exe** when errorlevel 12 was encountered, the **abcd.exe** program would change the errorlevel to a new value after abcd terminated. Since the batch file is executed one line at a time, the following errorlevel checks (from 11 through 1) would be testing the errorlevel set by **abcd.exe**, not the errorlevel set by Maximus!

To work around this limitation, you must use the **goto** statement for each errorlevel check. The goto statement allows your batch file to jump to a completely different location within the same batch file when a certain errorlevel is encountered. An errorlevel-based goto statement looks like this:

```
if errorlevel erl goto label
```

As before, *erl* is the errorlevel value to be tested.

The *label* value is a unique, alphanumeric, single-word name that indicates the destination of the jump. (Examples of valid label names are "GotCaller," "Did-ScanBld" and "Recycle.")

In English, the above statement reads:

If the *errorlevel* value is greater or equal to the value specified by *erl*, jump to the label in the batch/command file specified by *label*.

To specify the destination of the jump, you must *declare* the label by placing the same name at another point in the same batch file. This lets the operating system know where it should jump to when it encounters the *goto* statement.

A label declaration looks like this:

```
:label
```

*label* is the same label name that was specified in the original *goto* statement. As soon as the command processor spots a statement of the form “*goto label*,” it will jump to the location marked with “*:label*”.

For example, the following sample batch file:

```
Line 1: :Top
Line 2: echo Diamonds are forever
Line 3: goto Top
```

causes the line “Diamonds are forever” to be repeated over and over on the screen.

When the operating system starts the batch file, it processes each line in sequence. After reading line 1, the OS recognizes that “Top” is simply a label definition, so it skips to the next line.

After reading line 2, it processes the **echo** statement and displays “Diamonds are forever.”

Finally, after reading line 3, it realizes that it has to jump to the label marked “Top.” Since the “Top” label is at the top of the file, it goes back to line 1 and repeats the entire process over and over again.

However, the *goto* statement does have practical applications. The previous *Maximus errorlevel* example could be rewritten like this:

```
max -w -p1 -b38400
if errorlevel 12 goto OpA
if errorlevel 11 goto OpB
if errorlevel 10 goto OpC
if errorlevel 9 goto OpD
if errorlevel 3 goto OpE
if errorlevel 1 goto OpF

:OpA
```

```
echo Do operation "A" here.  
goto End  
  
:OpB  
echo Do operation "B" here.  
goto End  
  
:OpC  
echo Do operation "C" here.  
goto End  
  
:OpD  
echo Do operation "D" here.  
goto End  
  
:OpE  
echo Do operation "E" here.  
goto End  
  
:OpF  
echo Do operation "F" here.  
goto End  
  
:End
```

In this situation, the OS first compares the errorlevel returned by Maximus to those listed in the “if errorlevel” portion of the batch file. When it finds a match for the errorlevel, it jumps to the corresponding label.

For example, if Maximus exited using errorlevel 10, the batch file interpreter would jump down to the “OpC” label and process the “echo Do operation ‘C’ here” statement. (Most of the time, you would run an actual program after checking for an errorlevel, rather than simply echoing a string back to the console.)

After processing the echo statement, the command processor reads and processes the next line of the batch file. The “goto End” statement ensures that the command processor skips over the following commands after the “OpD” label definition. (Recall that the command processor simply ignores label definitions. Without the extra “goto End,” the batch file would just “fall through” to the statements under the OpD and OpE labels. The extra goto statement specifically instructs the command processor to jump to the “End” label at the end of the file.)

However, it may also be desirable to have the batch file “fall through” a set of errorlevels in certain cases. This allows errorlevels to be defined such that a certain errorlevel value, such as the “user entering EchoMail” value, also executes the command associated with another errorlevel value, such as the “user entered NetMail” value.

### 3.9.5. Sample WFC Batch File

Maximus uses errorlevels 1 through 4 for internal purposes, but errorlevel values of 5 and greater can be configured by the user. A standard Maximus installation uses the following errorlevel values:

*Errorlevel 255.* Maximus terminated with an undefined error condition. Your batch file should return to your front-end mailer or restart Maximus in WFC mode.

*Errorlevel 12.* A user entered EchoMail (and possibly also NetMail) during this session. In response, your batch file should call an external program to export mail to the network. If you are using the Squish mail processor, this command should be “squish in out squash”. In addition, if you use any \*.MSG format message areas, you must also call SCANBLD at this point. Finally, after calling all of these external programs, your batch file should return to your mailer or restart Maximus in WFC mode.

*Errorlevel 11.* A user entered NetMail (but no EchoMail) during this session. In response, your batch file should call your mail packer to export mail to the network. If you are using the Squish mail processor, this command should be “squish squash”. In addition, if you use any \*.MSG format message areas, you must also call SCANBLD at this point. Finally, after calling all of these external programs, your batch file should return to your mailer or restart Maximus in WFC mode.

*Errorlevel 5.* A user logged off and entered neither EchoMail nor NetMail. If you use any \*.MSG format message areas, you must call SCANBLD at this point. Your batch file should then return to your mailer or restart Maximus in WFC mode.

*Errorlevels 4 and 3.* A non-fatal error occurred. Your batch file should return to your mailer (or restart Maximus in WFC mode) if either of these errorlevels are detected.

*Errorlevel 2.* The caller hung up before entering a valid name at the log-on prompt. Your batch file should return to your mailer or restart Maximus in WFC mode.

*Errorlevel 1.* The SysOp pressed <alt-x> at the main WFC screen. Your batch file should exit back to the operating system.

The following **runbbs.bat** (or **runbbs.cmd** for OS/2) can be used to start Maximus in WFC mode and accept callers:

#### DOS only!

```
@echo off
rem * DOS users only:
rem *
rem * This line loads your FOSSIL driver.

bnu
```



**OS/2 only!**

```

rem * OS/2 users only:
rem *
rem * Comment out the above call to BNU
rem * and uncomment the following MODE command.
rem *(This command should be all be on one line.)
rem *
rem * mode com1:19200,n,8,1,TO=OFF,XON=ON,
rem *      IDSR=OFF,ODSR=OFF,OCTS=ON,DTR=ON,RTS=HS

:Loop
max -w -p1 -b19200
if errorlevel 255 goto Error
if errorlevel 16 goto Error
if errorlevel 12 goto EchoMail
if errorlevel 11 goto NetMail
if errorlevel 5 goto Aftercall
if errorlevel 4 goto Error
if errorlevel 3 goto Error
if errorlevel 2 goto Loop
if errorlevel 1 goto Done

:EchoMail
rem * Invoke scanner and packer here. Next,
rem * go to the "Aftercall" label to process
rem * any after-caller actions.
rem *
rem * For the Squish mail processor, use the
rem * following command:
rem
rem SQUISH OUT SQUASH -fc:\max\echotoss.log

goto Aftercall

:NetMail
rem * Invoke packer here, then go to
rem * the "Aftercall" label.
rem
rem For the Squish mail processor, use the
rem following command:
rem
rem SQUISH SQUASH

goto Aftercall

:Aftercall
rem * Invoke after-each-caller utilities here.

```

```

scanbld all
goto End

:Error
rem * Something bad happened, so let's say so.

echo An error occurred!
goto Down

:End
rem * This label should re-load your phone
rem * answering program. If you are using
rem * the Maximus WFC, you want to jump back
rem * to the top of the loop:

goto Loop

:Down

rem * The system arrives here if there was a
rem * problem.

echo Error! Maximus had a fatal error and
echo could not continue!

:Done
exit

```

To start Maximus, simply run the above **runbbs.bat** from the command prompt. Maximus will initialize and accept as many incoming connections as required.

For sample batch files that demonstrate how to use Maximus with a front-end mailer, please see Appendix G.

### 3.10. Step 10: Miscellaneous Information

You have now completed the installation procedure for Maximus. Although Maximus is now mostly installed, please keep these things in mind:

*For users of \*.MSG areas.* A renumbering utility is required. If you carry any EchoMail areas with a lot of traffic, a renumbering utility will be especially crucial. Maximus is bundled with the MR program. MR automatically deletes, renumbers and links messages based on information given in the message area control file. For more information on MR, please see section 8.9.

*For users of Squish areas.* Although Squish normally renumbers areas messages are created, you may occasionally need to use the SQPACK utility. SQPACK compacts

message area data files by removing unused space between messages. Most systems will only need to run SQPACK once a week, but it may be beneficial to run SQPACK on a daily basis for systems with a lot of traffic. SQPACK is part of the Squish product, the companion mail processor for Maximus. The Squish product also includes a number of other useful utilities, including a diagnostic and repair utility for Squish areas.

Your Maximus system should now be capable of handling callers, but many other options and features can be customized. The following section describes how to maintain some of the major components in a Maximus system.



---

# 4. Customization

This section describes how to customize the main components of a Maximus system. This section is just an overview of the possible customizations. For a full list of features and/or control file keywords, please see section 18.

## 4.1. Events and Yelling

The distribution copy of Maximus comes with a preconfigured *event file*. This event file serves two purposes:

- With the internal WFC subsystem, the event file is used to schedule “external events.” External events are used for running external programs at predefined times. These events are useful for performing daily system maintenance, general cleanup, and anything else you may require.
- The event file also controls *yell events*. Yell events are used to define the times of the day when callers are allowed to page the SysOp. A yell event also controls how many times the SysOp can be paged and the tune to be played during the page.

All events are kept in the file called **events##.bbs**, where **##** is the node number of the task (in hexadecimal). For a single-line system, **##** will be “01”.

Each node on a multinode system must have a separate events file. However, all of the event files use the same format, so you can simply copy a master events file to **events01.bbs**, **events02.bbs**, and so on. (If Maximus cannot find the event file for a specific node, it will try to read default event information from **events00.bbs**, regardless of the node number of the current session.)

### 4.1.1. Yell Events

The distribution version of Maximus comes with an event file called **events00.bbs**. The standard event file contains a single yell event that looks like this:

```
Event All 13:00 23:00 bells=3 maxyell=3 tune=random
```

This yell event is active between 13:00 and 23:00. (This means that the user is allowed to page the SysOp between 1 pm and 11 pm.)

If desired, additional yelling time slots can be added by simply duplicating the “Event” line and changing the starting and ending times.

The **bells** flag indicates the number of times that the bell or tune will be played.

The **maxyell** flag indicates that a user is allowed to yell up to three times during one session before the SysOp page feature is automatically disabled. (If the SysOp enters chat with the user, the yell count for that session is reset.)

Finally, the **tune** flag indicates the tune to be played during the page. Maximus includes a large library of tunes in the `\max\tunes.bbs` file. Specifying “tune=random” instructs Maximus to play a tune at random. However, you can also specify an explicit tune name, such as “tune=DigitalPhone”.

#### 4.1.2. External Events

The event file also supports *external events*. An external event causes Maximus to exit with an errorlevel at a predefined time of day or on a certain day of the week. Events are only active when Maximus is started in WFC mode.

To add an external event, simply add a line to `events##.bbs` with the appropriate starting time, and add an “exit=**level**” flag to the end of the line.

**level** specifies the errorlevel that Maximus will exit with when the event time occurs.

After creating an entry for the event in the `events##.bbs` file, you should modify your `runbbs.bat` file to handle the specified errorlevel and run the appropriate external command.

Please see section 18.11 for more information on the events file.

#### 4.1.3. SoundBlaster Support

**OS/2 only!** This section applies only to OS/2 users.

Maximus-OS/2 includes internal support for the SoundBlaster and SoundBlaster-compatible sound cards.

When playing yell tunes, Maximus will automatically determine whether or not a SoundBlaster is installed. If a SoundBlaster is found, Maximus will play the tunes from `tunes.bbs` on the SoundBlaster, rather than making noises on the internal PC speaker.

However, the SoundBlaster detection is not completely automatic. You must include the following line in your **config.sys** file to enable SoundBlaster support:

```
SET OS2BLASTER=Abase Iirq Ddma NOCLI
```

where **base** is the card's base address, **irq** is the card's IRQ level, and **dma** is the card's DMA channel. The "NOCLI" parameter must be included literally. This parameter tells the SoundBlaster code not to steal the CPU for long periods of time.

Except for the last "NOCLI" flag, the OS2BLASTER settings are identical to the settings for the standard DOS "BLASTER" environment variable.

The settings for a standard SoundBlaster card are:

```
SET OS2BLASTER=A220 I7 D1 NOCLI
```

## 4.2. Access Control: Classes, Privilege Levels and Locks

This section describes the access control and security mechanisms supported by Maximus 3.0.

### 4.2.1. User Classes

Maximus uses the *class* concept to control access to menu options, message areas, and other system components. A class describes the access rights and privileges of a group of users. The distribution version of Maximus includes 12 predefined user classes, but additional classes can be defined by the SysOp.

A class definition typically includes the following attributes (among many others):

- maximum time limit,
- maximum number of calls per day,
- maximum daily download limit,
- file download:upload ratio,
- the name of a special display file to be shown at log-on, and
- special system settings, such as the ability to write messages in read-only areas, unlimited on-line time, the ability to download any file on the system, and more.

(A complete list of class attributes can be found in section 18.9.)

In addition, classes are associated with specific *privilege levels*. A privilege level is a numeric value from 0 to 65535. The privilege level is generally proportional to the level of access granted to users in that class.

To make classes easier to manage, Maximus also assigns a name to each class. You can refer to classes either by name or by privilege level.

The classes included in the distribution version of Maximus are listed below in Table 4.1:

**Table 4.1 Standard Privilege Levels**

| Class Name | Privilege Level |
|------------|-----------------|
| Hidden     | 65535           |
| SysOp      | 100             |
| AsstSysOp  | 90              |
| Clerk      | 80              |
| Extra      | 70              |
| Favored    | 60              |
| Privil     | 50              |
| Worthy     | 40              |
| Normal     | 30              |
| Limited    | 20              |
| Demoted    | 10              |
| Transient  | 0               |

The class names and privilege levels are completely configurable, so if desired, the above names can be changed to reflect the different types of users on your system.

#### **4.2.2. Privilege Levels**

In the system user file, Maximus assigns a specific privilege level to each user. Since Maximus stores only the numeric privilege level, this means that you are free to rename a class or change its attributes at any time. (The privilege level assigned to new users is controlled by the **Logon Level** feature in the system control file.)

When a user logs on, Maximus compares the user's privilege level to the level of all defined user classes. Maximus then selects the class with the privilege level that is closest to (but not over) the user's privilege level.

For example, assume that a user is assigned a privilege level of 20. When the user logs on, Maximus will scan the class definitions and notice that the "Demoted" class has a privilege level of 20. This is an exact match of the user's privilege level, so Maximus will treat the user as a member of the Demoted class. The user will inherit all of the characteristics of the class, including the associated time and file download limits.

By separating the access limits from the user record, it becomes easy to adjust the permissions for a broad group of users by modifying a single class definition. Since



the user record stores only the class's privilege level, you can easily rename the class (or replace it with another one) without modifying the user file.

In addition, a user's privilege level does not need to exactly match one of the class privilege levels. Assume that a user is assigned a privilege level of 45. Maximus will select the user class with the level that is closest to (but which does not exceed) the user's privilege level. Given these criteria, Maximus will select the "Worthy" class (which has a privilege level of 40).

Although the user's privilege level is slightly higher than the Worthy privilege level, the user is still considered to be a member of the Worthy class. The user assumes the same access restrictions (including time limits and download limits) as all other members of the Worthy class.

From this, we see that all possible privilege levels (from 0 to 65535) can be converted into a specific user class. If a class's privilege level is defined to be  $x$ , and if the privilege level of the next-highest class is defined to be  $y$ , all privilege levels in the range  $x$  to  $y - 1$  (inclusive) are considered to be part of the first class.

This means that a user class actually encompasses a range of privilege levels. Assuming the standard class definitions given above, the standard privilege levels fall into the classes shown in Table 4.2:

**Table 4.2 Privilege Level Ranges**

| Class Name | Privilege Range |
|------------|-----------------|
| Hidden     | 65535           |
| SysOp      | 100 - 65534     |
| AsstSysOp  | 90 - 99         |
| Clerk      | 80 - 89         |
| Extra      | 70 - 79         |
| Favored    | 60 - 69         |
| Privil     | 50 - 59         |
| Worthy     | 40 - 49         |
| Normal     | 30 - 39         |
| Limited    | 20 - 29         |
| Demoted    | 10 - 19         |
| Transient  | 0 - 9           |

Now, since all users in a given class are granted the same access rights, you may wonder why Maximus allows multiple privilege levels to be assigned to a class. The answer lies in how Maximus processes access control definitions for commands and menu options: the access level for a menu command can be defined in terms of a class *or* in terms of a specific numeric privilege level.

For example, although all members of the Worthy class (levels 40 through 49) have the same time limit and download restrictions, the SysOp can define a menu option

that requires a privilege level of 46, which means that only some of the members of the Worthy class will be able to access the option.

From this, one can see that the 12 standard user classes can be logically subdivided into many more individual access levels. And should the original 12 classes not be enough to fit your needs, more classes can be added as necessary.

In situations where Maximus needs to display a user's privilege level, such as on the status line at the bottom of the screen, Maximus will first examine the class records to see if the user's privilege level is an exact match for one of the classes. If so, Maximus will display the class name. Otherwise, if Maximus cannot find an exact match, it will display the numeric privilege level.

#### **4.2.3. SysOp and Hidden Attributes**

Of the 12 classes defined by Maximus, only two classes have extraordinary attributes:

Users in the SysOp class have access to all system features and functions, including the ability to read private messages, modify users in the system user editor, and examine any file on the local hard drive.

Users in the Hidden class have no access to the system whatsoever. If a user's privilege level is changed so that it falls into the Hidden class, Maximus will immediately hang up when the user calls. A user can be placed into the Hidden class to lock that user out of the system.

In addition, menu options can be assigned an access level of Hidden. A "Hidden" menu option cannot be accessed by any user on the system, regardless of the user's privilege level.

Aside from the Hidden and SysOp classes, most of the other classes have only minor variations in time and download limits, so those classes can be assigned to normal users.

#### **4.2.4. Locks and Keys**

In addition to the user classes and privilege levels described above, Maximus supports a set of *keys* that can be assigned to each user. A key is equivalent to a "yes/no" flag that can be turned on or off on a user-by-user basis.

Maximus supports a total of 32 keys. Keys are referenced by a single letter or number. The 32 keys consist of the letters A through X and the numbers 1 through 8. Any or all of these keys can be assigned to users on an individual basis.

Keys are independent of a user's privilege level or class. For example, a user with a privilege level of 20 (in the "Limited" class) could have keys *I*, *3*, *D* and *L*. A user with a privilege level of 15 (in the "Demoted" class) could have keys *3*, *4*, *A* and *L*.

Keys are useful when some commands or menu options are only made available to a certain subset of users, regardless of privilege level. To restrict access to a system feature, the SysOp can "lock" the feature using a certain key or set of keys.

For example, a company BBS could have several different file areas dedicated to different products. Owners of a certain product could be given key *A*, while owners of a different product could be given key *B*. The file areas could be restricted so that a user needs key *A* to access information related to the first type of product, while the other file area could be restricted to those with key *B*. If a user happened to own both products, the user could be given both keys *A* and *B* to permit access to both types of areas.

Similarly, an area could be restricted to users with keys *A*, *B* and *C*, such that only users who owned all three types of products would be able to access the area.

Keys are also independent of a user's privilege level, so you can still assign different time and download limits to different classes of users, regardless of which keys are assigned to each user.

#### 4.2.5. Access Control Strings

Maximus uses an *Access Control String* (ACS) to describe the access requirements for menu options, message and file areas, and various other system components. An ACS can restrict a certain feature based on a user's class, privilege level, key settings, and numerous other attributes.

The simplest form of an ACS is simply a privilege level or the name of a user class. For example, if a specific menu option is assigned an ACS of:

46

then only users with a privilege level of 46 or above would be able to access that option.

Similarly, if you list the name of a user class, Maximus will convert the class name into a privilege level and compare it against the user's privilege level. For example, if a specific menu option is assigned this ACS:

Privil

then only users with a privilege level of 50 or above would be able to access that option. (This example assumes that your class definitions assign a privilege level of 50 to the “Privil” class.)

In addition to the names of the defined user classes, an ACS of **NoAccess** indicates that access is not granted to any user. The **NoAccess** string may be useful if you have removed the “Hidden” user class.

An ACS can also be used to ensure that only users with a certain set of keys are allowed to access an option. To add a key restriction, simply append a “/” to the end of the privilege level or class name, and then list the keys that the user must possess to access the command.

For example, this ACS restricts a command to users who have a privilege level of at least 55 and who have keys 2, 3, 5 and A:

```
50/235A
```

The name of a user class can also be used anywhere that a privilege level definition can be used, so the above ACS could be restated as:

```
Normal/235A
```

An ACS can also restrict a command to users who do *not* have a certain set of keys. To add such a restriction, simply insert a “!” in front of the key that you wish to negate. For example, the following ACS:

```
Worthy/23!6A!C
```

restricts a command to users who:

- have a privilege level of at least 40 (assuming the standard class definitions),
- have keys 2, 3 and A, and
- have *neither* key 6 nor key C.

All of the above ACS examples have restricted a feature to users with a privilege level that was *greater than or equal to* a certain value. An ACS can also restrict a command to users with an exact privilege level, or perform other types of logical tests.

To add a logical test to an ACS, insert one of the operators shown below in Table 4.3:

**Table 4.3 Access Control String Operators**

| Operator | Description   |
|----------|---|
| =        | Grants access if the user’s privilege level is exactly the specified level. |

|          |  |
|----------|--|
| >        | Grants access if the user's privilege level is strictly greater than the specified level.              |
| <        | Grants access if the user's privilege level is strictly less than the specified level.                 |
| >=       | Grants access if the user's privilege level is greater than or equal to the specified level (default). |
| <=       | Grants access if the user's privilege level is less than or equal to the specified level.              |
| <> or != | Grants access if the user's privilege level is not equal to the specified level.                       |

---

For example, the following ACS:

```
<=Normal/123
```

restricts access to those users who have a privilege level of 50 or less and who also have keys 1, 2 and 3. Assuming the standard user classes, this could also be restated as:

```
<=30/123
```

In addition, an ACS can restrict an option to a user with a specific name or alias. The following ACS restricts a feature so that only the user named "John Doe" is able to access it:

```
name=John_Doe
```

Notice that the space in the user's name is replaced with an underscore. An ACS may not include any spaces.

Similarly, the following ACS restricts a command so that only the user with an alias of "Peter Rabbit" can access it:

```
alias=Peter_Rabbit
```

Finally, boolean operators can also be included in an ACS definition. You can combine two existing ACSs by inserting the *and* operator ("&") between them:

```
10/123&<=Normal
```

This ACS restricts a command to users who have a privilege level of at least 10 but less than 30 (the "Normal" privilege level). Users must also have keys 1, 2 and 3 to access this command.

Similarly, the logical *or* operator can also be inserted between two ACSs:

```
<=Normal/12|AsstSysop/!J
```

This ACS restricts a command to users who either:

- have a privilege level of 30 (“Normal”) or less and have keys 1 and 2, *or*
- have a privilege level of at least 90 (“AsstSysOp”) and do *not* have key J.

Most major Maximus subsystems use the ACS concept to restrict access to features. For example, every message and file area can be assigned an ACS to control which users get access to that area, and many settings in the system control file also accept an ACS. However, a small number of features can only be controlled on the basis of privilege level or user class.

### 4.3. Display Files: \*.mec and \*.bbs

All of the information files that Maximus displays to the user are stored in the **.bbs** and **.mec** file formats. Collectively, these are known as *MECCA* files or display files. Most of the system MECCA files are stored in the `\max\misc` and `\max\hlp` directories, but you can add your own display files in other places.

The names of many of the standard display files can be found in the system control file, but the names of some display files cannot be changed. Please see Appendix H for more information on the names of these display files.

You will notice that most of these files come in pairs: for every file with a **.mec** extension, there is always a file with a **.bbs** extension. Just like control files, MECCA files must be compiled before they can be used by Maximus.

The **.mec** file is the source for a display file. You can edit the **.mec** file with a text editor to insert commands and display text. After you have finished modifying the **.mec** file, the MECCA compiler must be run to convert it to a **.bbs** file.

Compiling a file with MECCA is easy. Simply type in the command “MECCA **filename**”, where **filename** is the name of the **.mec** file to be compiled. For example, to compile the file **applic.mec** into **applic.bbs**, enter the following at a command prompt:

```
cd \max\misc
mecca applic
```

MECCA source files contain plain text to be displayed to the user, but they can also contain *tokens* to perform color changes, cursor control, conditionally display certain lines, and display system information. A MECCA token is a special keyword that is enclosed inside a pair of square brackets.

For example, a **.mec** file that contains the following:

```
[white>Hello there, [user].
Are you having a nice [lightblue]day [white]today?
```

will display “Hello there, John Doe” in white (assuming that the user’s name is John Doe). It will then display “Are you having a nice” in white, the word “day” in blue, and the word “today?” in white.

MECCA supports many other tokens that display information to the user or change screen attributes. MECCA allows you to embed user-specific or system-specific information into any display screen.

To see an actual MECCA file, load the `\max\misc\newuser2.mec` file with an ASCII editor. As you can see, the file consists mainly of ASCII text, but a few special MECCA tokens have been inserted to colorize the screen and perform other actions.

One of the main advantages of using MECCA is that only one set of display files needs to be created. Unlike other bulletin board packages where the SysOp must create both an ASCII and an ANSI version of a specific display file, Maximus automatically filters out color and graphics codes for those users who do not support ANSI or AVATAR graphics.

For compatibility reasons, Maximus comes with a utility to convert files containing ANSI graphics into MECCA files. Please see section 8.2 for more information.

Although MECCA files are normally viewed by running Maximus and displaying the menu option or command that contains the display file, you can also use the ORACLE utility to display a MECCA file from the command prompt.

For more information on creating MECCA files, please see section 17.

For more information on using the MECCA compiler, please see section 8.8.

For more information on ORACLE, please see section 8.10.

## 4.4. Message Areas and File Areas

The next step in customizing your system is to set up the message and file sections. The Maximus distribution kit comes preconfigured with a set of sample message and file areas, but most SysOps will want to customize these areas.

All message areas are defined in the **msgarea.ctl** file. Likewise, all file areas are defined in the **filearea.ctl** file.

In Maximus, each area (whether a message area or a file area) must have a unique name. Area names can be up to 64 characters long, and names can include both letters and numbers. Maximus supports a theoretically unlimited number of message and file areas, but it is better to start with a small number of areas and expand them as your system grows.

#### 4.4.1. Message Area Definitions

A message area definition looks like this:

```
MsgArea name
    % Other message area definition keywords
    % go here.
End MsgArea
```

where **name** is the name of the message area to be defined. All of the keywords related to that area must be placed between the **MsgArea** keyword and the following **End MsgArea** keyword.

Title: Aff  
 Creator:

By default, all log-off comments are placed in message area 1. If you wish to change the **name** for area 1, you must also change the **Comment Area** definition in the system control file.

Some common message area definition keywords are described below. A complete list of keywords can be found in section 18.6.

##### *ACS string*

Restrict access to this area so that only those users who satisfy the **ACS string** are allowed to see or enter the area. This statement is required for all message areas.

##### *Desc text*

Use **text** as the description for this area. Maximus will display this description when the user requests a list of areas.

##### *Path filename*

**filename** specifies the physical disk file and/or directory to use for storing messages.

For a Squish-format message area (default), **filename** must contain the path and filename of the area (without an extension).



For a \*.MSG-format message area, **filename** must contain only the path of the area. (Only one \*.MSG area can be stored in any given directory.)

### *Style flags*

The **flags** option specifies a number of optional flags and toggles for the message area. Multiple flags can be specified for a single message area by separating flag names with spaces. Table 4.4 describes some of the more common flags:

**Table 4.4 Message Style Flags**

| Flag   | Description   |
|--------|---|
| Pvt    | Allow private messages in this area. Private messages can only be read by the SysOp, the sender, and the addressee. |
| Pub    | Allow public messages in this area. Public messages can be read by anyone.  |
| Squish | Store this area in the Squish format (default).   |
| *.MSG  | Store this area in the *.MSG format.  |

Most of these flags are optional. Please see section 18.6 for information on other supported styles.

### *Tag name*

This keyword tells Maximus to use **name** as the “tag” for this area. If a tag is assigned to an area, Maximus will write the tag out to the **Log EchoMail** filename after the user logs off. This feature is normally used in conjunction with EchoMail areas; the tag specified here should be the same as the tag that you have defined for that area in your EchoMail processor.

#### **4.4.2. File Area Definitions**

A file area definition looks like this:

```
FileArea name
    % Other file area definition keywords
    % go here.
End MsgArea
```

where **name** is the name of the file area to be defined. All of the keywords related to that area must be placed between the **FileArea** keyword and the following **End FileArea** keyword.

Some common file area definition keywords are described below. A complete list of keywords can be found in section 18.7.

#### *ACS string*

Restrict access to this area so that only those users who satisfy the ACS **string** are allowed to see or enter the area. This statement is required for all message areas.

#### *Desc text*

Use **text** as the description for this area. Maximus will display this description when the user requests a list of areas.

#### *Download path*

This keyword tells Maximus where to find the files contained within this file area. The files that the users are to download must be contained within this directory.

#### *Upload path*

This keyword tells Maximus where to place uploaded files. There are two options for defining an upload path:

- Set the upload path to point to the same directory as the download path. With this configuration, files will be made available to other users as soon as they are uploaded. (However, users must ensure that they change to the correct file area before uploading files.)
- Set the upload path in *all* file areas to point to one common directory. This directory can then be configured as the *download* path for an area that can be accessed only by the SysOp.

This option is the most secure, since it allows the SysOp to check uploaded files before they are put on-line for other users. Users will only be able to see the file after the SysOp has checked the file and used the **Hurl** command to move it to another file area.

#### 4.4.3. Custom Message and File Area Menus

By default, Maximus will dynamically generate a menu when users select the **Area Change** command from the message or file area menus. This display can be controlled to an extent, using the **Format MsgFormat** and **Format FileFormat** definitions, but these commands may still be too restrictive for some SysOps.

Consequently, Maximus allows you to completely disable the automatic menu generation feature and replace it with custom **.mec** screens. The **Uses MsgAreas** and **Uses FileAreas** statements (in the system control file) instruct Maximus to display the specified file instead of generating an area menu of its own.

While these files give you a large degree of flexibility, using a custom display file means that you must remember to modify the display file when you add or remove an area.

#### 4.4.4. Message Area and File Area Hierarchies

Maximus allows you to group your message and file areas into logical, multi-level hierarchies. By default, all message and file areas are stored in a “flat” name space, but you can add **divisions** to your message area and file area control files to group areas in a logical manner.

For example, if your system supported the following message areas:

```
Lexus
Lawnmowers
C
BMW
Pascal
Shovels
Jaguar
```

the areas could be grouped in a more logical structure as follows:

```
cars.lexus
cars.bmw
cars.jaguar
programming.c
programming.pascal
garden.lawnmowers
garden.shovels
```

By adding the appropriate **MsgAreaDivision** records to your message area control file, Maximus can automatically insert the “cars.” or “programming.” prefix for each area. Maximus will also present the list of areas to the user in a logical manner. For example, the top level message area menu might look something like this:

```

Message Areas -----
cars          ... DIVISION: Cars
programming   ... DIVISION: Langs
garden        ... DIVISION: Gardening

```

If a user selected “cars” at the prompt, Maximus would display a submenu of all of the areas in that division, including *cars.lexus*, *cars.bmw* and *cars.jaguar*.

Maximus also supports nested message area divisions. For example, the “cars” division could be subdivided into two separate groups of areas, such as “cars.luxury” and “cars.economy.”

All of the comments related to message areas also apply equally to file areas. The description below refers to message area divisions only, but you can apply the same concept to file areas by replacing **MsgDivisionBegin** with **FileDivisionBegin**, **MsgDivisionEnd** with **FileDivisionEnd**, and **MsgArea** with **FileArea**.

To create a message area division, you must place the following statement *before* the definition of the first message area to be included in the division:

```
MsgDivisionBegin name acs display_file description
```

In addition, you must place the following statement *after* the end of the last message area to be included:

```
MsgDivisionEnd
```

The **name** parameter is the name of the message area division. From the example above, one might use a name of “cars” or “garden.” Maximus will automatically add this name, followed by a period (“.”), to the names of any message areas defined within this division.

The **acs** parameter is the ACS that controls access to the message area division. A user can only see this area division if the user’s privilege level passes the access control check.



The **acs** parameter only controls the access level required for users to *see* the division on the message area menu. Even if users cannot see a message area division, if they know the name of an area inside the division, and if they have a privilege level high enough to access the message area (but not the division), the users can change directly to that area anyway. For this reason, the **ACS** of a message area definition should be at least as restrictive as the ACS of the **MsgDivisionBegin** statement.

The **display\_file** parameter instructs Maximus to display the specified file when the user requests a list of areas in the division, rather than automatically generating a menu of its own. This display file is normally used in conjunction with the **Uses**

**MsgAreas** definition in the system control file. If you do not wish to use a custom display file, specify a “.” to instruct Maximus to generate the area list automatically.

The **description** parameter is a short description of the contents of the division. Maximus will display this description on the message area menu when the user requests a list of areas.

For example, to implement the message division structure described in the previous example, use the following syntax:

```
MsgDivisionBegin cars    Demoted . DIVISION: Cars
  MsgArea lexus
    % Definitions for the Lexus area go here
  End MsgArea

  MsgArea bmw
    % Definitions for the BMW area go here
  End MsgArea

  MsgArea jaguar
    % Definitions for the Jaguar area go here
  End MsgArea
MsgDivisionEnd

MsgDivisionBegin programming Demoted . DIVISION: Langs
  MsgArea c
    % Definitions for the C area go here
  End MsgArea

  MsgArea pascal
    % Definitions for the Pascal area go here
  End MsgArea
MsgDivisionEnd

MsgDivisionBegin garden Demoted . Gardening
  MsgArea lawnmowers
    % Definitions for the lawnmowers area go here
  End MsgArea

  MsgArea shovels
    % Definitions for the shovels area go here
  End MsgArea
MsgDivisionEnd
```

## 4.5. Maintaining File Areas

Message areas can be created quite easily, but file areas require a fair amount of maintenance to run properly. Not only do you need to create file area definitions, but you must also create listings of files which can be downloaded by the user.

(However, if you wish to create a blank file area, no extra work is required, since Maximus will create a file listing as soon as a user uploads a file to that area.)

#### 4.5.1. Creating File Listings

To place files in a newly-created file area, you must create an ASCII listing of the files in the area. If you already have a directory containing the files to be added, you should first copy those files to the directory specified in the file area's **Download** statement.

Next, from the command prompt, change the current directory to the directory specified in the area's **Download** path. Next, enter the following command to create the initial file catalog:

```
for %f in (*.*) do echo %f >>files.bbs
```

If you wish to run this command from a batch file, instead of from the command prompt, enter this instead:

```
for %%f in (*.*) do echo %%f >>files.bbs
```

After a bit of disk activity, the file listing should be created for you. Although this will create a listing of file names, most users will also want to see file descriptions. To add file descriptions, run an ASCII editor (such as the DOS **edit.com** or the OS/2 **e.exe**) and load the **files.bbs** file.

Inside **files.bbs**, you should see a list of the files in the directory. If you wish to add a description for a particular file, simply add one or more spaces after the filename and insert your description there. A description can be up to 1024 characters long; although only 48 characters can be displayed on one line of the file catalog, Maximus will automatically wordwrap the rest of the text onto the following lines.

The **files.bbs** in a hypothetical file area could look like this:

```
DEMO.LST      Description of product demonstration
INFO.TXT      Information about this system
RESULTS.ZIP   Results from the last quarter
```

To add files to the file listing after performing the initial "for %f" command, you can simply use a text editor to insert a line in **files.bbs** and add the name and description for the file.

Similarly, to delete a file from the listing, just remove the line containing the file entry from **files.bbs**. You should also delete the actual file from the download directory.

When using the default **File Date Automatic** setting in the system control file, Maximus will automatically colorize the listing and place the file's size and date beside the filename.

In addition, Maximus allows files to be marked as “free downloads”. Files can be marked for “free download bytes” (the file does not count against the user's download limit) or “free download time” (the file does not count against the user's time limit), or both.

To mark a file as a free download, you must add a slash sequence before a file's description. For a free time download, insert “/t” before the description. For a free bytes download, insert “/b” before the description. For both free time and free bytes, use “/tb”.

For example, to ensure that the Maximus 3.0 distribution files do not count against the user's download quota, use the following:

```
MAX300C.ZIP /b This is Maximus 3.0.
```

If you want to count the download against the user's byte total (but not the user's time limit), use the following:

```
MAX300C.ZIP /t This is Maximus 3.0.
```

Similarly, if you want both free time and bytes to be given to the user, use the following:

```
MAX300P.ZIP /tb This is Maximus 3.0.
```

Finally, you can add comments to **files.bbs** which are not specifically related to a file. If the first character on a line is a dash (“-”) or a space (“ ”), Maximus will treat the line as a comment and display it to the user. In addition, if the first character on the line is a dash, the line will be displayed in white. If the first character is a space, Maximus will display the line in the current color (which is normally cyan).

#### 4.5.2. Global Downloading, Fast Locates and Duplicate Files

The *global downloading* feature allows users to select a file for download from any point in the system, regardless of the file area where the file is stored. For example, if your system had a file in area “os2” called **fix.zip**, a user could enter “d fix.zip” from file area “dos” and still download the correct file.

The *fast locate* feature allows Maximus to perform very fast file area searches using the **Locate** command.

The *duplicate checking* (or *dupe check*) feature allows Maximus to compare the filenames of uploaded files with the files that are available for download. Maximus can be configured to automatically reject files that already exist elsewhere in the file areas.

To enable any of these features, you must use the FB utility to create a compiled file database. FB scans the **files.bbs** catalogs in all file areas and creates a binary file database. This database is required for all of the features mentioned above.

Every time you change your file areas, you should run the following command:

```
fb -a
```

This command tells FB to compile a database containing all of the file areas defined in **filearea.ctl**. (On large systems, this command may take a long time to execute. For information on incremental FB compiles, please see section 8.5.)

After running FB, the global downloading and fast locate features are automatically enabled. However, you may need to enable the **Upload Check Dupe** feature in the system control file to use duplicate file checking.

#### 4.5.3. CD-ROM Handling

To set up your system to retrieve files from a CD-ROM, you must first create a **FileArea** definition for each directory on the CD. However, to inform Maximus that the area is stored on slow media, you must add the following keyword to the file area definition:

```
Type CD
```

This line tells Maximus to do two things:

- Maximus will only access the actual drive when absolutely necessary. For example, this means that Maximus will not try to validate the directory when displaying an area list.
- Files to be downloaded from this area will be copied to a staging area on your hard drive before the transfer. This helps prevent thrashing when multiple users are downloading from a rotary CD changer. (The destination directory for this copy operation can be set with the **Stage Path** definition in the system control file.)
- SILT will not check to see whether or not the directory exists. This means that you can compile your system with support for many different CDs, even if your system only has one CD-ROM drive.



**DOS only!**

In addition, when setting up your system with a CD, do not forget to edit the **Save Directories** keyword in the system control file. This keyword tells the DOS version of Maximus to save the current directory for every drive in a list of drives. However, since CD-ROM drives have removable media, you must remove the drive letter corresponding to the CD-ROM from the **Save Directories** statement.

**4.5.3.1. File Listings**

Some CD-ROMs may already be set up for use with a Maximus system. Such CD-ROMs will come with a **files.bbs** file in every directory that contains files to be downloaded. If this is the case, you do not need to do anything further.

However, a number of other CD-ROMs are not set up for BBS use (or do not have a standard Maximus **files.bbs** listing). If this is the case, you need to construct your own listing of files in each directory.

To construct your own file listing for one area, add the following line to the file area definition:

```
FileList    c:\path\filename.bbs
```

The **FileList** keyword instructs Maximus to look for a **files.bbs**-like catalog with the specified name. The CD-ROM is read-only, so you need to point **filename** to a file on your hard drive.

In this file, you must place a **files.bbs**-like listing of files in the area. The file name should come first, followed by the file description, just as in **files.bbs**.

When a user accesses the area, instead of looking in the **Download** path for the **files.bbs** file, Maximus will look for the file list specified in the **FileList** statement.

For example, a sample CD-ROM area definition might look like this:

```
FileArea apl
  Type      CD
  Desc      The APL Programming Language
  Download  g:\msdos\apl
  Upload    c:\max\file\upload
  FileList  c:\max\lists\apl.bbs
End FileArea
```

In this case, the **c:\max\lists\apl.bbs** file would be a file catalog listing all of the files in the **g:\msdos\apl** directory.

### 4.5.3.2. Listing Date and Size Formats

Maximus supports three different styles of “file dating.” These formats are:

*Automatic dating.* Maximus retrieves a file’s date and size from the directory entry. This is the default style. When FB builds the file database, and when a user does a file listing for a single area, Maximus retrieves file information from the download path.

*List dating.* Maximus assumes that a file’s date and size are stored as part of the file description in the **files.bbs** listing (or the **FileList** catalog for the area). FB will read the file description and parse dates and sizes in the form “mm-dd-yy size” or “size mm-dd-yy.” FB will incorporate this information into the compiled file database. Maximus will also colorize the file information when displaying a directory listing. When list dating is used, the download directory is not examined at all.

*Manual dating.* Maximus does not attempt to process or display file dates or sizes in any manner whatsoever. The **File\_NewFiles** option and the “L\*” (or “F\*”) commands cannot be used to search based on file dates when this dating method is selected.

The **Type** keywords shown below in Table 4.5 can be used to specify one of these dating styles.

**Table 4.5 File Area Date Styles**

| Keyword         | Style                 |
|-----------------|-----------------------|
| Type DateAuto   | Automatic file dates. |
| Type DateList   | List dating.          |
| Type DateManual | Manual file dates.    |

The **DateAuto** style is well suited for file areas stored on your hard drive. Aside from file areas that never change, this is usually the best option to use.

However, the **DateList** style is much more appropriate for CD-ROMs. If the CD comes with a file catalog that contains file sizes and dates, select the **DateList** style and use the **FileList** keyword to specify the name of the file containing the file list.

For example, if the CD-ROM vendor has placed a **DateList**-compatible file list in the download directory (called **00index.txt**), the following file area definition can be used:

```
FileArea apl
  Type      CD DateList
  Desc      The APL Programming Language
  Download  g:\msdos\apl
  Upload    c:\max\file\upload
  FileList  g:\msdos\apl\00index.txt
```

End FileArea

If you use a **DateList** file area, Maximus will not examine the drive at all when the user requests a file listing. In addition, the CD-ROM does not need to be in the drive when FB is asked to build the file database. For CD-ROMs, the **DateList** option is the quickest in terms of overall system performance, and it is much easier to maintain than the other two options.

#### 4.5.3.3. CD-ROM areas and DateAuto Processing

If you still choose to use **DateAuto** processing with a CD-ROM file area, you must follow the following procedure to maintain the file catalog.

Normally, since **DateAuto** processing reads information from the file download directories, all of your file areas must be accessible when you run FB. However, if you have only one CD-ROM drive and many CDs, you may wish to create file areas for all of your CDs, even though only one CD can be on-line at a time.

Using the **DateList** option is preferable when you want to do this. However, if you would like to use **DateAuto** processing, you must follow this procedure when building your file catalog with FB:

1. Remove all CDs from the computer and run “fb -a”. This compiles the file database information for file areas on your hard drive.
2. Insert the first CD into the drive. Run FB only over those areas which are contained on that CD. (If you have subdivided your file areas using **FileDivision** statements, you can tell FB to compile only a certain file area tree using wildcards, such as with “fb simtel.\*”. Otherwise, you will have to specify each area separately on the command line like this: “fb area1 area2 area3”.)
3. Remove the first CD and repeat step 2 for all remaining CDs.
4. After completing step 2 for all of your CDs, the file area database will be completely built. However, to prevent FB from trying to recompile CD areas when the CD-ROM is not in the drive, you **must** always run FB with the “fb -s” parameter (which causes it to skip the CD areas). After the file database is built, if you omit the -s parameter even once, the compiled file information for your CD-ROMs may be overwritten.

## 4.6. Barricades and Extended Barricade Files

The **Barricade** keyword in the message and file areas tells Maximus that you wish to protect the area with a password, or that you want to grant different access levels to certain users while they are in that area.

When a user enters a barricaded area, the **Uses Barricade** file is shown, and the user is given three tries to enter the correct access code.

The access codes required to enter a barricaded area are contained in a *barricade file*. The **Barricade** keyword in the area definition must point to a barricade file.

#### 4.6.1. Barricade Files

A barricade file is a straight ASCII text file containing a list of passwords, with each password followed by the privilege level to grant to users who enter the correct password.

Each line of the barricade file is in the following format:

```
<password> <priv>
```

**<password>** is the password which grants a privilege level of **<priv>** to the user. **<priv>** can only specify a numeric privilege level or a class name. (Keys are not valid in the **<priv>** string.)

For example:

```
helloworld SysOp
kentucky Clerk
cleanse Normal
scum Demoted
```

If a user enters an area with this barricade file and types “helloworld” at the password prompt, the user will be granted access to the area, and the user’s privilege level will be set to SysOp while in the area.

Similarly, if the user typed “kentucky,” the user’s privilege level would be temporarily altered to Clerk while inside that area.

In addition, if you specify “NoAccess” in the **<priv>** field, a user who enters the specified password will be denied access and told that the area does not exist.

#### 4.6.2. Extended Barricade Files

Maximus supports the “extended” barricade file concept. An extended barricade file allows users to be promoted without using passwords.

Before displaying the **Uses Barricade** file, Maximus will quickly scan the barricade file to see if it uses the extended barricade syntax. If so, Maximus skips the **Uses Barricade** display and processes the barricade file directly.

Each line in an extended barricade file has the following format:

```
!<user_name>  <priv>
    or
!All          [priv]
```

The “!” in the first column of each line is not optional. The “!” distinguishes an extended barricade file from a normal barricade file.

<user\_name> is the name of the user whose access level is to be promoted. No spaces can be present in <user\_name>, so replace spaces with underscores. (For example, to use Joe SysOp in an extended barricade, you would have to use “Joe\_SysOp” for the <user\_name>.)

If Maximus matches the name for the user trying to enter the barricaded area, the user's privilege level is altered to <priv> with no questions asked.

In addition, if the user's name is not explicitly found in the barricade file, Maximus will “fall through” to the optional “!All” keyword.

If you use “!All” by itself, without specifying a privilege level, Maximus will let other users into the area using their real privilege levels. If you do specify a privilege level after the “!All” keyword, Maximus will let all users enter the area and promote all of them to a level of [priv].

The “!All” statement must be at the very end of the barricade file to function properly.

Finally, you can even use the “NoAccess” privilege level with extended barricades. This allows you to create a barricaded area that is completely invisible to certain users.

For example:

```
!Jesse_Hollington  Clerk
!Steven_Bonisteel  Clerk
!Hubert_Lai        Privil
!All               Demoted
```

This file assigns the privilege level of “Clerk” to the first two users, assigns the “Privil” level to the third user, and gives all other users a privilege level of “Demoted”. (To prevent anyone except the above three users from accessing the area, the “Demoted” could be replaced with a “NoAccess”.)

## 4.7. Menus

This section describes how to configure different parts of the Maximus menu system.

### 4.7.1. Customizing Menu Options

The system menus are completely redefinable and offer a great deal of flexibility. However, if you are just starting a new Maximus system, it is best to skip making major menu changes until your system is up and running.

However, even novices can change the access levels required to access particular commands in the menus control file. The access control string (ACS) for each menu option is normally located in the second or third column of each menu option. You can easily change these ACS definitions to suit your own system's privilege level or user class scheme.

You can also modify the "Command as it appears to the user" field with a certain degree of safety. This command is what is shown on the screen when Novice-level menus are active.

However, when changing menu commands, ensure that the *first* character in the command is unique among all options on that menu. When the user enters a letter at a menu, Maximus will process all commands that begin with that letter. Hence, if you use a letter that has already been used by another command, confusing things will happen when a user gets to that menu.

### 4.7.2. Custom Menu Display Files

Maximus allows you to create custom menus with relative ease. Instead of displaying the standard yellow and gray menu, Maximus can be configured to display a user-defined **.mec** or **.bbs** file.

To add a custom menu file, simply insert a **MenuFile** keyword at the top of the appropriate menu definition in **menus.ctl**.

Some of the following tips may be helpful when creating custom menus:

- When using a custom menu that contains the *[cls]* token, the output from some of the internal commands (such as the **Version** command) *may* disappear, since the *[cls]* token in the menu erases it before it can be seen.

To solve this problem, you must *link* a **Press\_Enter** menu option after the appropriate command. This will cause Maximus to wait until the user presses

`<enter>` before redisplaying the menu. For more details, please see section 4.7.3.

For example, to make Maximus wait after displaying the version screen, you can use something like this:

```
Version                               Demoted "Version"
NoDsp Press_Enter                     Demoted "V"
```

This method is described in more detail in the following section.

- When designing a custom menu with an input prompt at the bottom, you may have some trouble getting the cursor to stop in the appropriate place. Most text editors automatically insert an `<enter>` after the last line of the file; since Maximus reads the entire file, this causes the cursor to skip down to the next line when the entire file is displayed.

Three possible solutions to this problem are:

1. Use a text editor that does not insert carriage returns at the end of files.
2. If you are using a `.mec` file to create the menu, insert a `[quit]` token where you want the cursor to stop. As soon as Maximus encounters this token, it will stop displaying the file without displaying the extra carriage return.
3. If you are creating the menu file manually, you can insert the compiled equivalent of the `[quit]` token directly into the `.bbs` file. The compiled equivalent is `"<ctrl-o>Q"`.

#### 4.7.3. Linking Menu Options

When working with custom menus, the output of some of internal Maximus menu options may occasionally not fit in with the layout of your menu. For example, if you are using a custom menu that always clears the screen, the output of some commands may disappear before the user has a chance to read it.

The solution to this problem is *menu option linking*.

When the user selects a key, Maximus will search the entire menu for a menu option that has a description starting with that key. When it finds one, it executes the specified option.

However, Maximus continues to search for other commands with that key, even after executing the first menu command. This means that a number of menu commands can be tied to a single keystroke. (To prevent the second and subsequent

menu options from appearing on the menu, use the **NoDsp** modifier in front of the menu option.)

Consider this scenario: a user selects a message area and begins to read messages. The standard message menu is quite large and it leaves little space on-screen for messages. However, if you include something like this on the message menu:

```

                Read_Next                Demoted "Next Msg"
NoDsp  Display_Menu  READMSG Demoted "N"
                Read_Previous            Demoted "Previous Msg"
NoDsp  Display_Menu  READMSG Demoted "P"
```

after the user selects a message with **Next** or **Prior**, Maximus will automatically display the READMSG menu. This menu can then display only a limited subset of commands that are useful when reading messages.

## 4.8. QWK Mail Packing

Maximus includes a built-in QWK mail facility for off-line mail reading. This feature allows callers to log on, pack up messages from one or more message areas, and download a compressed mail bundle for off-line reading and reply. The packer is fully integrated with the rest of the BBS, and the packer will automatically adjust itself as message areas are added to or deleted from your system.

All of the QWK-specific information is stored in the reader control file. To customize the off-line reader, you should edit a copy of **reader.ctl** with a text editor and make the following modifications.

- You must give a unique name (of up to 8 characters) for the **Packet Name** keyword. This keyword is used when creating QWK packets to send to your users. (Do not include the **.qwk** extension.) Try to make this name describe your BBS in some way, such as an abbreviation truncated to eight characters.
- You should also modify the **Phone Number** keyword to reflect your system's true phone number. Maximus does not use this information, but users who download QWK packets will receive this phone number along with their mail.
- You may also want to customize the **Max Messages** keyword. If you run a busy system and wish to restrict callers from downloading more than 200 messages at a time, you can set a maximum value here. In the distribution control file, users are prevented from downloading more than 1200 messages at a time. To completely disable this limit, comment out the **Max Messages** keyword.

Beyond this initial configuration, the QWK packer is completely self-maintaining. No extra maintenance is required. However, if you would like to add extra features



to the off-line mail packets, such as bulletin listings and new file lists, please see section 5.3.

## 4.9. Multilingual Support

Maximus 3.0 is fully multilingual. Up to eight different language files can be defined in the system language control file, and users can switch between languages at any time (using the **Chg\_Language** option on the Change Setup menu).

The language files contain all of the text strings that Maximus sends to the user, including prompts, system messages and command keys. A separate language file can be created to use “Oui” and “Non” instead of “Yes” and “No”; even the key-strokes for various internal options can be changed.

Language files are divided into two distinct sections. Each language file has a set of strings to be displayed to the *user*, and each also has a second set of strings to be displayed to the *SysOp*. By default, the SysOp always sees the text strings contained in the *first* language file defined in **language.ctl**, regardless of the language selected by the user.

This means that the user can go through a set of menus in German, but the SysOp will still be able to read the pop-up menus and the system log file in English.

Maximus's multilingual support can be used to define different prompts, menus and custom display files for each individual language. Language files can be modified by simply editing the appropriate **<langname>.mad** file with a text editor.

However, a separate set of menus needs to be designed by the SysOp, since the screen display would look odd if the prompts were in German but the menu options were in English. Likewise, display files should also be changed (using the *[iflang]* token) to accommodate new languages.

The main method for supporting alternate menus and display files is to use the “%Y” external program translation character. When used in menu and display filenames, the “%Y” sequence translates to the user's current language number, with 0 being the *first* language defined in **language.ctl**, 1 being the second language, and so on.

The “%Y” sequence can be used in many places, including in the **First Menu** definition in the system control file, in all **Display\_Menu** options. In addition, the text “+Y” can be used in all **Display\_File** commands. (Note that **Display\_File** requires a “plus-Y” instead of a “percent-Y”.)

For example, if you had the following language statements in **language.ctl**:

## 82 4. Customization

```
Language English
Language Deutsch
```

using a **First Menu MAIN %Y** statement in the system control file tells Maximus to display MAIN0 for English callers, while MAIN1 will be displayed to German callers.

You can also use this methodology for MECCA files. You can either use a **Display\_File D:\Path\File%Y** option to display different physical files for each language, or you can embed the *[iflang]* token within an individual display file to make decisions based on the current language.

By default, Maximus stores a user's language preference in the user file. However, if you want Maximus to prompt the user for a new language during every log-on, you can place the *[menu\_cmd chg\_language]* token at the top of **welcome.mec**.

---

# 5. Maximus Subsystems

## 5.1. Waiting for Callers Subsystem

Maximus includes internal support for a Waiting for Caller (WFC) subsystem. This subsystem allows Maximus to initialize the modem, wait for a call, answer the phone, and pass control to the main portion of the BBS. The WFC subsystem can be used on all nodes of a system, on selected nodes, or on no nodes. Nodes which do not use the WFC subsystem require an external “front end” program to answer the phone.

### 5.1.1. Starting WFC

When starting Maximus, the WFC subsystem is enabled using the “-w” command line switch. Optionally, the “-p” and “-b” parameters can be used to override the COM port and baud rate. If you specify just “-w”, WFC starts up using the port and speed defined in the system control file.

Before enabling WFC mode, you must ensure that the modem strings in the system control file are configured correctly. The distribution version of Maximus is configured to support most Hayes-compatible modems. However, if the WFC module does not seem to work, you may need to modify certain definitions (such as the **Answer** and **Init** strings) to make it perform as expected.

In particular, the distribution version of Maximus attempts to use “manual answer mode.” Instead of telling the modem to automatically answer the phone when it detects a ring, Maximus will perform the ring checking on its own. This is the preferred method of operation, since the phone is only answered when Maximus is ready to accept a call.

However, manual phone answering may not be compatible with all modems. If you wish to disable manual answering, change the last part of the **Init** string to read “S0=1” instead of “S0=0”. You must also comment out the **Answer** string. This instructs your modem to answer the phone automatically.

### 5.1.2. Screen Display and SysOp Keys

After WFC mode initializes, four multicolored windows appear on the screen:

The first window, “Status,” displays the time until the next system event, the current modem status, the number of calls made to your system (both today and in total), and the name of the last caller on your system.

The second window, “Modem Responses,” displays a scrolling list of responses from the modem. Maximus uses this window for storing result codes that are sent in response to command strings. (Maximus will automatically filter out any “OK” results, so only meaningful responses will be displayed.)

The third window, “Current Activity,” displays system log messages as they occur.

The fourth window, “SysOp Keys,” contains descriptions for all of the keys that can be pressed while in WFC mode.

Pressing `<alt-k>` starts Maximus in local mode. Maximus takes the phone off-hook and commences the normal log-on procedure.

Pressing `<alt-j>` invokes a shell to the DOS or OS/2 command interpreter. Maximus takes the modem off-hook while you are in the shell. You can perform file maintenance, make changes to your batch files, or do other routine operations while in the shell. Type “exit” to return to Maximus.

Pressing `<alt-x>` instructs Maximus to take the system down. Maximus will put the phone off-hook, clear the screen, and exit to your batch file with errorlevel 1.

When using the internal WFC subsystem, Maximus also supports *external events*. External events can be used to run a particular program at a certain time of day, normally by exiting to your batch file with a certain errorlevel.

For more information on external events, please see section 18.11. For more information on installing WFC, please see section 3.9.

## 5.2. Local File Attaches

### 5.2.1. Description

Maximus allows users to create *local file attaches*. A local file attach is a file that is associated with a message created by a user.

To enable local file attaches, ensure that you have defined an **Attach Path** keyword in **max.ctl**. This keyword tells Maximus where to store uploaded files.

Next, add the **Style Attach** attribute to Squish-format message areas that you want to use for local file attaches. (The local file attach feature cannot be used with \*.MSG areas.)

To create a file attach, users simply change to that message area and enter a message. When the cursor is on the attributes field in the message header (which is where the “Pvt” flag is normally displayed), the user can enable the “file attach” flag. (In the English version of Maximus, the A key is used to create a file attach.)

After setting the flag, the user can continue to fill out the message header and create the message itself. After the message has been saved, Maximus will prompt the user to upload the file to be attached to the message.

When the transfer is complete, Maximus will compress the attached file (using the archiver specified by **Attach Archiver** in the system control file) and store it in the file attach directory.

Later, when the addressee displays the message, the user is given the option to download the file attach. Maximus will then decompress the file and send it to the user.

In addition, a user can use the **Msg\_Download\_Attach** menu option to display a list of all unreceived files attached to that user.

### 5.2.2. SysOp Configuration

The local file attach subsystem is mostly self-maintaining. Using the **Kill Attach** keyword in the system control file, you can configure Maximus to automatically delete files after they have been received by users.

The **Message Edit Ask LocalAttach** keyword can also be used to control the privilege level required to create a local file attach.

If you expect to receive a lot of file attaches in one particular message area, the holding area for the attaches can be overridden on an area-by-area basis using the **AttachPath** keyword in your message area control file.

In addition, the `[msg_fileattach]` MECCA token can be used to conditionally display text if any unreceived file attaches are waiting for the current user.

Finally, Maximus also supports inbound FTS-0001 style “NetMail file attaches” in areas that have both **Style Net** and **Style Attach**.

When Maximus encounters a message in a NetMail area with the network file attach bit set, it looks in the directory specified by **Path Inbound** in the system control file. In that directory, it tries to find a file with the name specified on the mes-

sage's subject line. If that file exists, Maximus treats it as a file attach and sends it to the user.



If you add the **Style Attach** attribute to a NetMail area, you may be compromising the security of your inbound directory. Do not use the local file attach feature in NetMail areas unless you trust all of the users who have access to that area.

## 5.3. QWK Mail Packer

From the Off-Line Reader menu, Maximus allows users to download mail to be read off-line. Maximus supports the popular QWK format, so the users can use popular third-party products such as Deluxe2, Qmail, SLMR and OFFLINE to read the packets generated by Maximus.

Instructions on configuring the QWK packer are given in section 4.8.

### 5.3.1. Bulletins, News Files and File Lists

QWK packets can include information other than just mail, such as bulletins, file lists and news files. Maximus supports these files in an extremely simple manner: any file placed in the `\max\olr` directory is automatically placed in all mail packets sent to users.

In addition, if a user has unreceived local file attaches, they are automatically placed in the QWK packet.

The QWK format defines the names of several standard files that can be included in QWK packets. To use these features, simply place a file in the `\max\olr` directory with one of the filenames given in Table 5.1:

**Table 5.1 QWK Packet Filenames**

| Filename | Description   |
|----------|---|
| hello    | Displayed when the off-line reader first starts up. This is typically the equivalent of your welcome.bbs screen. This file should be ANSI only; no AVATAR colors or MECCA codes are allowed.                      |
| news     | Your BBS news file, equivalent to the <code>\max\misc\bulletin.bbs</code> file. This is usually available as an option from the QWK reader's main menu. This is normally a flat ASCII file with no graphics.      |
| goodbye  | Displayed when the reader closes the packet from your BBS. This file can include ANSI graphics.   |
| blt-1.1  | Bulletin file 1. This is usually displayed as an option on the reader's main menu. In this case, the file extension is ".1", but you can use anything from ".1" to ".99" to provide up to 99 different bulletins. |

|              |  |
|--------------|--|
| newfiles.dat | This file contains a new files listing for your BBS. Maximus does not generate this file for you, but a third-party file list generator can be configured to generate a file list and place it in the <b>\max\olr</b> directory. |
|--------------|--|

---

Again, all of these files are optional. However, since Maximus packs up everything in the **\max\olr** directory when creating a packet for the remote system, simply placing one of the above files in that directory will cause it to be displayed on the remote side.

### 5.3.2. Message Packing for Remote Users

The default Maximus configuration includes an Off-Line Reader menu, but mail can also be packed from the regular message menu. All of the QWK mail packing functionality is built into the **Browse** command; in fact, the **Download** command on the Off-Line Reader menu is a simple macro that invokes the **Browse** command.

The default **Download** command passes the options *t n p* to Browse. This requests a scan of **tagged** areas, searching for **new** messages, and for the messages to be **packed** in QWK format. Obviously, the flexibility of the Browse command allows many other search operations to be performed. Users can specify a selective download by using the search function (complete with **and** and **or** operators). Using Browse, messages can also be packed only from the current area, rather than all tagged message areas.

After selecting the **Pack** option from the Browse menu, Maximus will gather all of the specified messages, display some statistics on the packed messages, and ask the user whether or not the messages should be prepared for download. If so, Maximus will compress the packet with the user's default archiving program, count to ten (giving the user a chance to abort), and send the file using the default transfer protocol.

The Off-Line Reader menu also includes an upload option. This option allows the user to upload a **.rep** file created by an off-line reader. The **.rep** file is automatically decompressed, regardless of the archive type or the user's default archiver, and the messages within are placed in the appropriate message area.

When processing uploaded messages, the QWK upload command always checks to ensure that the user has enough access to write a message in a given message area. To do this, it first examines the definition for the target message area. If the area has a custom **MenuName** defined, Maximus will read that menu. Otherwise, Maximus will read the menu called "MESSAGE".

Next, Maximus will search that menu for an option of type **Msg\_Upload**. Maximus checks the ACS for that option and compares it against the user's access level. If the check succeeds, the user is allowed to upload the message.

This access control mechanism has one main implication: unless you use a specific **MenuName** definition for all of your message areas, you *must* have a menu with a name of “MESSAGE,” and that area *must* have a **Msg\_Upload** option with an ACS that makes it accessible to users.

### 5.3.3. Local Mail Packing

The QWK feature can also be used by local callers. After compressing a packet for a local user, Maximus will leave the packed QWK file in the off-line reader directory. (By default, the file will be in the `\max\olr\node##` directory, where `##` is the current task number in hexadecimal.)

In local mode, if you want to “upload” a **.rep** packet, select the **Upload** option from the reader menu. If the caller is local, Maximus will prompt for the path and filename of the **.rep** packet. Enter the location of the packet (as created by your off-line reader), and Maximus will decompress and import the messages in that packet.

### 5.3.4. Unattended Mail Packing

In conjunction with the “-j” command line parameter, Maximus can compress mail packets for users on a daily basis, without operator or user intervention.

At predefined intervals, you can set up a system event (when running in WFC mode) to call Maximus with the “-j” command line parameter. The -j parameter tells Maximus to insert a list of keystrokes in the keyboard buffer, as if the keystrokes were entered by a local user. You can set up these keystrokes so that Maximus will log on as a certain user, execute a download command, log off, and have your batch files copy the created QWK packet to a file area.

By doing this, you can pack mail for certain users in advance, or you can use it to save mail for yourself while on vacation. Since the packing process is completely controlled by the keystrokes you specify for the -j parameter, almost any type of mail download is possible.

For example, suppose that the following keystrokes are required to move from the bulletin menu (displayed just after the user enters a password) to the off-line reader menu, download a packet, and log off:

```
n;o;d;i;y;m;g
```

To instruct Maximus to log on as a specific user and execute these keystrokes, you only need to add the prologue that specifies the user name and password. Consequently, the following command sequence can be used to automatically pack mail for a user, assuming the default menu and bulletin file structure.



```
max "-jJoe User;y;Password;n;o;d;y;m;g"
```

Title: Af  
Creator:

If your log-on sequence includes a “Press ENTER to continue” prompt, you should specify the “|” character where you would normally press the `<enter>` key.

In addition, you can pack mail for multiple users by simply replicating the above line in your batch file. However, your batch file must also copy the QWK packet from the `\max\olr\node##` directory into a safe place after each mail pack.

### 5.3.5. NetMail Messages

The QWK format was not designed with NetMail messages in mind, so users must follow a special convention when reading and replying to NetMail messages. When downloading messages from a NetMail area, the first line of each message will look like this:

```
From: <addr>
```

where `<addr>` is the network address of the message sender. Since there is no place in the QWK header to store the true message origination address, Maximus places that information in the message body instead.

When creating or replying to a NetMail message, Maximus expects to find a “To: `<addr>`” line as the *first* line in the message body. For example, to send a NetMail message to the address 1:123/456, the first line of the message must look like this:

```
To: 1:123/456
```

The “To:” header will be stripped before the message is written to the Maximus message base, so your QWK messages will look like normal messages to everyone else.

When replying to a message, there is an easy way to set the destination address; simply quote the original message and change the “From:” line to a “To:” line (after removing any quoting marks). This ensures that the destination address is correct, and Maximus will ensure that your reply is sent to the intended destination.

## 5.4. RIPscrip Support

Maximus includes internal support for *RIPscrip* graphics commands. *RIPscrip* is a terminal protocol designed by TeleGrafix Communications, Inc. The *RIPscrip* protocol includes facilities for displaying buttons, icons and various other forms of graphics over a serial line. *RIPscrip* also allows the remote user to use a mouse to access system commands and features.

The distribution version of Maximus comes with support for *RIPscrip* graphics, including a number of sample screens, but this support must be enabled before it can be selected by users. To enable *RIPscrip* graphics support, set the **Min RIP Baud** definition (in the system control file) to the minimum speed that you require for users to be able to view *RIPscrip* graphics. (In most cases, setting this to 2400 or 9600 is usually a good idea.)

After *RIPscrip* support is enabled, users will be able to select *RIPscrip* graphics using the **Chg\_RIP** command on the Change Setup menu. In addition, new users will be prompted for *RIPscrip* graphics support when they log on.

Maximus does not display *RIPscrip* graphics on the SysOp screen. (In fact, the local output routines include a “smart” filter that automatically strip out *RIPscrip* graphics sequences from local output.)

However, Maximus includes the following *RIPscrip*-specific features:

- When instructed to display a **.bbs** file, Maximus will first look for a file with a **.rbs** extension. To add a *RIPscrip*-specific version of one of your system display files, simply create the *RIPscrip* file using the same base name as the standard display file, but use a **.rbs** extension instead of **.bbs**.

To create **.rbs** files, you have one of two options:

1. Use a third-party *RIPscrip* editor to create the **.rbs** file directly.
  2. Use MECCA to compile a **.mer** file into a **.rbs** file. In addition to the standard **.mec** extension, MECCA also recognizes **.mer** files as containing MECCA source. You can embed MECCA commands inside *RIPscrip* graphics in this manner.
- If you decide not to create separate **.rbs** files, small *RIPscrip* sequences can still be included in standard display files. To mark a section of a MECCA file so that it is only displayed to callers with *RIPscrip* support, use the *[rip]* and *[endrip]* tokens around the *RIPscrip*-specific text.
  - Similarly, the *[norip]* and *[endrip]* tokens can be used to mark a section of a MECCA file that is only displayed to callers who do *not* support *RIPscrip* graphics.
  - With *RIPscrip* enabled, many of the system prompts are displayed using *RIPscrip*-specific buttons and features.
  - An alternate set of *RIPscrip* strings are used in the language file. The standard **english.mad** language file is configured to return different display strings for some system display fields if the user supports *RIPscrip* graphics. This allows

the SysOp to define one set of responses for text-based callers and a second set of responses for *RIPscrip* callers.

- The full-screen editor and full-screen reader are *RIPscrip*-aware. The message header is drawn using *RIPscrip* commands, and the header remains on the screen even when entering the message editor.
- Maximus automatically parses the “set text window” and “set font” *RIPscrip* sequences. It uses this information to adjust the user’s virtual terminal size, control the “More [Y,n,=]?” prompts, size the full-screen editor, and so on.
- A number of *RIPscrip*-specific display files are included with the standard distribution.
- Maximus can automatically send *RIPscrip* scene and icon files to the remote user. The MECCA *[ripsend]* and *[riphasfile]* tokens can be used to conditionally send a set of scene or icon files to the remote user. The equivalent MEX functions, **rip\_send** and **rip\_hasfile**, can also be used for this task.
- When a user logs on, if that user’s profile indicates that *RIPscrip* graphics are supported, Maximus will automatically send a query to the user’s terminal program. If the terminal program does not report that it supports *RIPscrip* graphics, Maximus will display a warning to the user and offer to disable *RIPscrip* support.
- Maximus keeps track of a “RIP path.” This path is where Maximus looks for icon and scene files that are referenced by the *[ripsend]* and *[ripdisplay]* tokens. This path can be changed using the *[rippath]* MECCA token. (To implement multilingual *RIPscrip* support, you may want to have a separate directory of *RIPscrip* files for each supported language.)
- For consistency, when *RIPscrip* graphics are selected, Maximus forces the user to enable hotkeys, the full-screen reader and screen clears.
- Menu options can be made available only to *RIPscrip* callers by using the **RIP** modifier in front of a menu option. Similarly, menu options can be made available only to callers *without* *RIPscrip* support by using the **NoRIP** modifier.
- If a user enables *RIPscrip* support after failing the internal *RIPscrip*-detection test, Maximus will note this in the system log file. When Maximus queries the remote system and fails to obtain an intelligent response, Maximus will retry the command up to three times before aborting. At this point, Maximus assumes that the caller enabled *RIPscrip* graphics in error, so *RIPscrip* graphics are automatically disabled and Maximus displays a warning to the user.

## 5.5. User Expiration and Subscriptions

Maximus includes an internal user subscription and expiry subsystem. Callers can be set to *expire* based on the current date, or also based on system time used (in minutes). When a caller expires, Maximus can optionally demote that caller to a lower privilege level, hang up, or delete the user's account.

To access the user subscription system, start up the Maximus user editor by running “max -uq”.

To set up a user subscription, first press the *E* key to select the expiry menu. Next, press *E* again to set the “expire by” method:

If you want the user to expire after a certain date, select *D*. If you want the user to expire after having used a certain number of minutes on the system, select *M*. If you want to disable the subscription system, select *N*.

Next, press *E* again to go back to the expiry menu, and press *A* to select an expiry action. This field controls how Maximus treats the user when the subscription expires. If you want Maximus to hang up and delete the user's account, select *A*. To have Maximus demote the user to a lower privilege level, select *D* and enter the new privilege level for the user. If you do not want Maximus to do anything when the subscription expires, select *N*.

Finally, press *E* one last time to go back to the expiry menu, and press *D* to select the date/time for the user expiration. If you previously selected *date* in the “expire by” field, enter the user's expiry date here. Otherwise, enter the number of on-line minutes to be credited to the user.

After setting up the expiry controls for a user, the subscription system is completely self-maintaining. When a user expires, the user's privilege level will be modified accordingly as soon as the user logs in.

In addition, when a user expires using the “by date” expiration method, the file `\max\misc\xpdate.bbs` is shown. Similarly, when a user expires due to running out of on-line minutes, Maximus will display the `\max\misc\xpertime.bbs` file.

## 5.6. Message Tracking System

### 5.6.1. Introduction

Maximus includes an internal Message Tracking System (MTS). This feature makes it easy for organizations to ensure that technical support queries are an-

swered on a timely basis, and it provides audit trails for technical support queries. MTS is only supported for message areas stored in the Squish message format.

Message areas can be designated as MTS areas, which means that Maximus will automatically place all messages entered into those areas in the MTS database. Whenever a message is created in an MTS area, Maximus will automatically set a default message priority, status, and message owner. These MTS fields are hidden from normal users, but they can be viewed and modified by technical support personnel.

The owner field is used to “assign” a message to a specific support representative. After the representative assumes ownership of a message, it becomes that representative’s responsibility to ensure that the message is handled appropriately.

MTS uses relational database links to store owner information, thereby making it easy to perform mass ownership transfers. For example, if a particular representative goes on vacation for a number of weeks, a single database entry can be changed by the MTS administrator to assign all of the representative’s messages to another user. The change can then be reversed when the original representative returns.

Maximus handles this by assigning a four-character “alias” to each representative (or in MTS terminology, to each moderator). A message is actually owned by an specific alias, which is in turn linked to a moderator (representative).

In terms of manipulating MTS data, the system works best when the moderator can log onto Maximus to read and reply to messages. After a moderator replies to a message, Maximus provides an option that allows the moderator to change the status of that message, if necessary. The **Track** menu option can also be used to modify the message’s owner and priority, manually insert messages in the tracking database, generate reports, and perform database administration functions.

MTS also works with off-line QWK mail readers. QWK support is implemented by placing a small template at the beginning of each tracked message in a **.qwk** packet. (This template is only placed in the QWK message if the user’s access level is at least equal to the privilege level specified by **Track View**.)

To modify the status of a message from a QWK reader, the off-line database moderator can simply quote that template in the reply message, marking an “X” in the appropriate box with a standard text editor. Maximus will automatically strip the template from the uploaded message and make the required changes to the message database.

MTS can also be used as a central tracking system for messages created on remote systems. Even if a message was entered on another system and transmitted as an EchoMail message, when that message arrives on a Maximus system that runs MTS, it will be entered in the database when the message is accessed or read by any user. (Distributed *tracking databases* are not supported, since all of the tracking

files must be stored on one system. However, tracking of distributed *message bases* is supported by placing all imported messages into the local tracking database.)

By default, Maximus will add a message to the tracking database if:

1. the message area is declared with the “Audit” keyword,
2. the message area has a default owner assigned, and
3. the user entering the message is not the default owner.

MTS also allows the moderators to generate reports based on tracking information. Reports can be generated on-line, written to a file, or even included in a QWK packet. The reports can be configured so that only a certain subset of tracked messages are displayed; for example, a moderator can easily request a report of all “Open” or “Working” messages, with a priority of “Urgent” or “Critical,” owned by a specific moderators, across all of the message areas on the system.

### 5.6.2. Information Stored by MTS

The Message Tracking System stores the following information for each message in its database:

- The message owner, stored as a four-character alias.
- The message status. The status field contains one of the following values:
  - ◆ New
  - ◆ Open
  - ◆ Working
  - ◆ Closed

This field allows the moderators and the database administrator to assess the current status of a problem report.

- The message priority This field contains one of the following priority levels:
  - ◆ Notify
  - ◆ Low
  - ◆ Normal
  - ◆ Urgent
  - ◆ Critical

This field can be used to assign a relative importance to a particular message.

- The message audit trail. The audit trail is actually stored as part of the message body. The **Msg\_Kludges** menu option can be used to toggle display of the audit trail.
- A message comment. This is also stored as part of the message body, but it can be examined or modified using the Modify command on the Track menu.

The tracking mechanism is completely transparent to the end user. The MTS data can be displayed in the message header when a message is shown, but this information is only available to the MTS moderators.

### 5.6.3. Configuration

This section describes the settings and definitions that are required to enable support for MTS:

To use MTS, the keywords listed in Table 5.2 must be present in the system control file. Please see section 18.2.4 for more information on these keywords:

**Table 5.2 MTS System Keywords**

| Keyword       | Description  |
|---------------|--|
| Track Base    | Specifies the base path and filename for the MTS database.   |
| Track View    | An ACS that controls who can view tracking information in messages.  |
| Track Modify  | An ACS that controls who is allowed to access the <b>Track/Admin</b> menu and modify tracking information. |
| Track Exclude | An optional file listing users whose messages are to be excluded from the tracking database.               |

To configure a message area to support MTS, the keywords listed in Table 5.3 must be added to the definition in the message area control file. Please see section 18.6 for more information on these keywords.

**Table 5.3 MTS Message Area Keywords**

| Keyword       | Description   |
|---------------|---|
| Style Audit   | Indicates that the area supports MTS messages.  |
| Owner <alias> | Indicates the alias of the default owner for the area. See the following section for information on creating moderator/alias links. |

#### 5.6.4. Using MTS

Administrators can access the MTS database using the **Msg\_Track** command (which is normally the “%” option on the message menu).

The main MTS menu contains the options shown below in Table 5.4:

**Table 5.4 MTS Main Menu**

| Command        | Description  |
|----------------|--|
| Insert         | Manually inserts a message into the MTS database       |
| Modify         | Modify the tracking information for a specific message |
| Report         | Display a report based on messages in the MTS database |
| Administration | Change area and owner links or remove messages.        |

##### 5.6.4.1. Insert

The **Insert** command allows a moderator to add an existing message to the tracking database. Normally, this command need not be used, since messages in MTS areas are automatically added to the tracking database when they are created.

The **Insert** command prompts the user to enter the message number of the message to add, plus the new owner for the message. The owner can be any of the existing moderators in the tracking database. The new message is assigned a priority of *normal* and a status of *new*.

##### 5.6.4.2. Modify

The **Modify** command allows a moderator to change the status of an existing message. This command allows moderators to modify the message’s owner, status, priority and message comment. All changes made to the message are added to the audit trail.

Every time a moderator replies to a message that he/she owns, Maximus will prompt the moderator for a new message status. Almost all status changes occur when a moderator replies to a message, so most moderators will not need to use this command on a frequent basis.

##### 5.6.4.3. Report

The **Report** command allows a moderator to generate a report of messages in the tracking database. The moderator can select a set of criteria for displaying information from the database, including message area, status, priority and owner.



The tracking report contains a list of each message that matches the specified criteria, including the tracking identifier of each message, the date it was placed in the tracking system, the message status, priority and owner, and the location of the message (area and message number).

#### 5.6.4.4. Administration

The administration menu contains options for tasks which do not normally need to be performed on a daily basis. Moderators must have a privilege level of at least **Track Modify** to access the administration menu.

*Owner/alias links.* This menu allows the owner/area links to be created or modified. In the tracking database, the message owner is recorded as a four-letter abbreviation. The owner/alias link is used to associate a specific moderator with an abbreviation.

Use the **Add** command to provide a four-character alias, followed by the name of the owner to associate with that alias.

Use the **Delete** option to remove an existing owner/alias link.

Use the **List** option to list existing owner/alias links.

*Area/owner links.* This menu allows the default area owner links to be modified. Although SILT will automatically update these entries based on the **Owner** keyword in the message area definition, this menu can be used to manually add, delete, or list the current owner/area links.

The area/owner links control the default owner for newly-created messages in the specified area. Unless an area has a default owner, messages entered in the area will not be added to the MTS database.

*Remove message.* This option removes a message from the MTS database. The user is prompted to enter a message number, and the message matching that number is then removed from the database.

#### 5.6.5. QWK Message Processing

In addition to manipulating tracking data while on-line, MTS can also be accessed through the QWK mail packer. When Maximus packs a message that is stored in the MTS database, it adds a special “tracking header” and “tracking footer” to the top and bottom of the message. A user’s privilege level must satisfy the **Track View** ACS in order to be able to see this information.

The first header line, **ACTRACK**, contains the MTS identifier for the message being downloaded. This is used as a unique identifier in the MTS database.

The second header, **STATUS**, contains the status of the message. The curly braces (“{ }”) indicate the current status of the message. The other fields have square brackets (“[ ]”) to indicate the other possible settings for the message status.

In the message trailer, the **OWNER** line contains the four-letter identifier for the owner of the message.

The priority line, **PRTY**, indicates the current priority of the message. As before, the braces indicate the current priority, and the square brackets indicate the possible priority selections.

The comment line is a one-line field that the moderators can use to assign a comment to each message. The comment can be as long as will fit inside the square brackets.

The QWK moderator can modify these headers and footers in a reply message, causing Maximus to change the status of the message or perform some other operation when the **.rep** packet is uploaded. To communicate a change back to Maximus, the moderator must reply to the message, using the off-line reader’s quoting feature to copy the tracking lines into the reply.

The **ACTRACK** line must always be quoted, since it gives Maximus the information that it needs to link the reply with the original message.

Following the **ACTRACK** line, any of the other tracking lines can be quoted (and in any order). To change the status or priority of a message, simply put an “x” inside the set of square brackets for the desired status/priority. Do not add or delete any characters from the line; simply quote it verbatim and use overtype mode for making any necessary modifications.

The owner and comment lines can be changed by overtyping in a similar manner.

If an “x” is placed in the “Discard Reply Text” checkbox, Maximus will throw away the reply message after processing the tracking database changes. This should be used if a moderator does not wish to post a message but still wants to make a change to the message status.

When saving a message to disk, Maximus will automatically strip out the tracking lines (and the preceding quote marks).

The tracking reply messages are uploaded normally, just like any other **.rep** packets. Upon upload, Maximus will display a few status lines indicating the changes that it is making to the tracking database.

#### 5.6.6. EchoMail support

Maximus also supports tracking of EchoMail messages. Messages can be imported into the tracking database by a standard EchoMail tosser, and Maximus will automatically add the messages to the tracking database as they are read by users.

Maximus will automatically add remotely-entered messages which meet the following criteria:

1. The message area is already configured to add locally-entered messages, including the **Style Audit** and **Owner** definitions.
2. The message does not contain an “ACGHOST” kludge line.
3. The “local” attribute is not set.
4. The “date written” field is not the same as the “date arrival” field.

#### 5.6.7. Audit Trails

Maximus keeps an audit trail for all messages in the tracking database. This audit trail takes the form of timestamped entries which are added to the bottom of each message. An entry is added to the audit trail when:

1. the message is entered in the database,
2. the message is removed from the database,
3. the message status is changed,
4. the message priority is changed,
5. the message owner is changed, or
6. the message comment is changed.

To view the auditing information on-line, use the “!” key (**Msg\_Kludges**) to toggle the display of auditing information and other kludge lines. Auditing information is also available to off-line reader users whose privilege level is at least that given by the **Message Show Ctl\_A** definition in the system control file.

#### 5.6.8. Import/Export Facilities

MTS provides facilities for exporting and importing the tracking database to a delimited ASCII file.

The TRACKEXP program is used to export the tracking database. It creates the files shown below in Table 5.5:

**Table 5.5 Tracking Export Data Files**

| Filename    | Description                            |
|-------------|--|
| trkmsg.dbf  | Message ID / status / owner / location |
| trkarea.dbf | Default area / owner links             |
| trkown.dbf  | Owner / alias links                    |

The format of **trkmsg.dbf** is given below. The system creates one line per message. All fields are enclosed in double quotes:

```
<id>,<owner>,<area>,<msgnum>,<status>,<priority>,<date>,<time>
```

**<id>** is the 16-character tracking ID for the message in question.

**<owner>** is the four-character alias for the owner of the message. This is used as a link to the **trkown.dbf** database.

**<area>** is the full area name in which the message is stored.

**<msgnum>** is the unique message identifier representing the tracked message.

**<status>** is the status of the message. (0=New; 1=Open; 2=Working; 3=Closed.)

**<priority>** is the priority of the message. (0=Notification; 1=Low; 2=Normal; 3=Urgent; 4=Critical.)

**<date>** is the actual date of the message, in mm/dd/yy format.

**<time>** is the actual time of the message, in hh:mm:ss format.

The format of **trkarea.dbf** is as follows:

```
<area>,<owner>
```

**<area>** is the full name of the message area.

**<owner>** is the four-character alias for the area's default owner. This is used as a link to the **trkown.dbf** database.

The format of **trkown.dbf** is as follows:

```
<owner>,<name>
```

**<owner>** is the four-character alias for the owner.

**<name>** is the full ASCII username of the owner.

TRACKIMP can also be used to import data from **.dbf** files back into the tracking database.

TRACKIMP reads files with the same names and same formats as generated by TRACKEXP. In addition, TRACKIMP will only import new messages; it will display an error message if you try to import a message with an **<id>** that already exists in the database.

Consequently, if you want to export the tracking database, make changes, and re-import the resulting information, you must delete the current tracking database before importing the **.dbf** files again. To do this, remove the **trk\*.db** and **trk\*.i??** files from the **\max** directory before starting the import.

#### **5.6.9. Limitations**

Only Squish message areas are supported by MTS. \*.MSG areas do not provide any facility for assigning a unique number to a message, so renumbering messages in a \*.MSG area would invalidate all of the links in the tracking database. For this reason, \*.MSG tracking is not supported.



---

# 6. External Programs

Maximus offers a large number of internal features, but occasionally, you may wish to execute external programs while a user is on-line. Maximus can run almost all types of external programs, from customized file transfer protocols to “door” programs written for other types of BBS software.

Maximus can be configured to run an external program from a menu option, from a MECCA file, or even from a MEX program.

## 6.1. Execution Methods

Maximus supports three primary methods for running external programs. You should use the execution method that corresponds to the type of program that you want to run.

Table 6.1 describes the benefits of the different execution methods. (When in doubt, use the “DOS” method.)

**Table 6.1 External Program Types**

| Type | Description  |
|------|--|
| DOS  | <p>This is the so-called “standard” execution method for both DOS and OS/2 systems. Maximus loads a second copy of the command processor (either <b>command.com</b> for DOS or <b>cmd.exe</b> for OS/2) and tells it to run the specified program.</p> <p>This is the only way to run a <b>.bat</b> or <b>.cmd</b> file as an external program. This is also the only way to execute internal command processor commands, such as DIR, TYPE, CHDIR, and so on.</p> <p>Maximus will automatically search for the program on the current path.</p> <p>In the DOS version, an extra copy of <b>command.com</b> must be loaded, which will increase the memory requirements for your program by roughly 5k.</p> <p>This is the method used by the <b>Xtern_DOS</b> menu option and the <i>[xtern_dos]</i> MECCA token. The MEX <b>shell</b> function can also be configured to perform a DOS exit.</p> |

**Run** This execution method is similar to the DOS method, except that the command interpreter is not loaded. In general, programs executed with the Run method will load slightly faster than with the DOS method.

However, you cannot load **.bat** or **.cmd** files with this method.

This is the method used by the **Xtern\_Run** menu option and the *[xtern\_run]* MECCA token. The MEX **shell** function can also be configured to perform a Run exit.

**Errorlevel** This exit type is only supported under DOS. This tells Maximus to exit completely from memory and return to the calling batch file or program.

This method is very slow because the entire Maximus image must be reloaded after the external program has finished executing. In addition, the transient portion of **command.com** must also be reloaded.

This is the method used by the **Xtern\_Erlvl** menu option and the *[xtern\_erlvl]* MECCA token.

See section 6.3 for more information on errorlevel batch files.

---

## 6.2. Swapping

**DOS only!** By enabling the **Swap** keyword in the system control file, Maximus can swap itself to EMS, XMS or disk when executing a program by the DOS or Run execution methods. When Maximus is swapped out, it will occupy less than 3k of conventional memory.

Swapping using the DOS or Run methods is generally preferred to using the Errorlevel method. Swapping is also much faster than using the Errorlevel method if your system has free EMS or XMS memory.

## 6.3. Errorlevel Batch Files

When exiting using the Errorlevel method, Maximus can create an optional batch file to pass command line parameters to an external program.

To create an errorlevel batch file, you must specify an errorlevel to be used for the exit, in addition to the name of the program to run (including any optional parame-



ters). Consequently, an errorlevel exit looks very similar to a DOS or Run exit, except that an errorlevel is added to the beginning of the command to be executed.

When Maximus encounters an argument after the errorlevel, it will write a file called **errorl##.bat** in \max directory, where ## is the current node number in hexadecimal. (In the case of node 0, the file is called **errorlvl.bat**.) Maximus will place the argument string specified in the **Xtern\_Erlvl** exit into that batch file.

After Maximus exits, your **runbbs.bat** file can trap the errorlevel and use a “call errorl##.bat” command to execute the external program.

For example, given the following menu command:

```
Xtern_Erlvl 65_Mydoor.Exe_/p%p_/b%b Demoted "Door"
```

When Maximus executes this menu option, it will create an **errorl##.bat** file (with ## replaced by the node number) that contains:

```
Mydoor.Exe /p1 /b9600
```

Your batch file should check for errorlevel 65, issue a “call errorl##.bat”, and then reload Maximus. The procedure for reloading Maximus after an errorlevel exit is described in the following section.

## 6.4. Restarting After Errorlevel Exit

After executing an external program using the Errorlevel exit method, Maximus can restart itself exactly where it left off. To users, it will appear as if Maximus had remained in memory the entire time.

This feature is made possible by the “-r” command line parameter. When Maximus is invoked using “-r”, it reads a file called **restar##.bbs** from the \max directory. The Errorlevel exit routines will write this file to disk just before Maximus executes the errorlevel command. The **restar##.bbs** file contains all of the information that Maximus needs to restart in its original state.

When restarting Maximus with -r, you may also need to specify a number of other command line parameters. In particular:

- If you are using any of the following parameters:

-p, -s, -n, or an explicit **.prm** name

then you must also include these parameters in your call to “max -r”.

For example, if your standard Maximus start-up command looks like this:

```
max system2 -w -p2 -n3
```

then the call to restart Maximus with the -r parameter must look like this:

```
max system2 -p2 -n3 -r
```



Never attempt to use an errorlevel exit before a new caller has reached the `\max\misc\newuser2.bbs` file. Maximus cannot perform a restart until it knows who the user is, and that means that the user must have entered a name, password, graphics selection, and so on.

This sample batch file utilizes errorlevel batch files and the restart option:

```
rem * These first "%1 %2 %3" parameters will be
rem * passed to the batch file by your mailer.
rem * However, they are not really important
rem * when dealing with errorlevel batch files, so
rem * we will just assume that they are correct.
rem *
rem * Also, ensure that the ":DoErlvl" label
REM * comes after the main "Max -b%1 ..." command.

max -b%1 -p%2 -t%3 -n2

:DoErlvl
if errorlevel 65 goto Outside
REM * [...more errorlevels here...]
if errorlevel 1 goto end
goto end

:Outside
REM * Ensure that the number after the "-n" parameter
REM * specifies the Maximus task number to use, if
REM * not the one specified in the control file.
REM *
REM * Finally, if you are using a non-zero task
REM * number, keep in mind that the filename that
REM * Maximus writes will be "ERRORLxx.BAT", where
REM * "xx" is the task number in hexadecimal.

call C:\Max\Errorl02.Bat
Max -r -n2
goto DoErlvl

:End
```

After you have created a batch file such as this, using errorlevel exits becomes just as easy as any of the other exit types. In MECCA, instead of using something of this format:

```
[xtern_run]D:\Path\Programe.Exe Arg1 Arg2
```

one could easily use an Errorlevel exit as shown below:

```
[xtern_erlvl]65 D:\Path\Programe.Exe Arg1 Arg2
```

As you can see, once you have added the errorlevel code to your batch files, adding new options requires only a minimal amount of work.

## 6.5. External Program Translation Characters

When specifying command line parameters for external programs, and also in certain MECCA tokens, Maximus can include information about the current user by using special *external program translation characters*.

A external program translation character consists of a percent sign and a single, case-sensitive letter or symbol. Maximus interprets the character following the percent sign and replaces it with the requested information.

Table 6.2 describes the external program translation characters supported by Maximus:

**Table 6.2 External Program Translation Characters**

| Character | Translation  |
|-----------|--|
| %!        | Embeds a newline in a string.  |
| %a        | The number of calls that the user has made to the system, prior to the current call.                                 |
| %A        | The user's first name, in uppercase.   |
| %b        | The user's baud rate. If the user is a local caller, this translates to "0".   |
| %B        | The user's last name in upper-case. If the user has no last name, this token translates into "NLN" ("No Last Name"). |
| %c        | The user's city.   |
| %C        | The response to the last [menu] MECCA token.   |
| %d        | The current message area name.   |
| %D        | The current file area name.  |
| %e        | The user's password.   |
| %E        | The user's screen length, in rows.   |
| %f        | The user's first name, in mixed case.  |
| %F        | Path to the current file area.   |
| %g        | Graphics mode. (0 = TTY; 1 = ANSI; 2 = AVATAR.)  |
| %G        | Daily download limit (in kilobytes).   |
| %h        | The user's phone number.   |
| %H        | Number of kilobytes downloaded today   |
| %I        | Total downloads (in kilobytes).  |

|    |  |
|----|--|
| %I | Total uploads (in kilobytes).  |
| %j | Minutes on-line for this call.   |
| %k | The current node number. ("0" means no node number.)   |
| %K | The current node number in hexadecimal format, padded with leading zero to make it two characters. ("00" for no task number.)  |
| %l | The user's last name in mixed case. If the user has no last name, this token translates into "NLN".  |
| %L | If the user is remote, this token translates into the string "-pX -bY", where X is the port number (1=COM1, 2=COM2, and so on) and Y is the user's baud rate.<br>If the user is local, this translates into a simple "-k". |
| %m | The name of the first file to transfer when invoking an external protocol.   |
| %M | Path to the current message area.  |
| %n | User's full name in mixed case.  |
| %N | The name of your BBS, as defined in the system control file.   |
| %o | The user's privilege level.  |
| %p | The current port number (0=COM1, 1=COM2, and so on).   |
| %P | The current port number (1=COM1, 2=COM2, and so on).   |
| %q | Path to the current message area (with no trailing backslash).   |
| %Q | Path to the current file area (with no trailing backslash).  |
| %r | The user's real name, if applicable.   |
| %R | All remaining stacked text in the keyboard buffer, as entered at the last menu.  |
| %s | The SysOp's last name in mixed case. If the SysOp has no last name, this translates into "NLN".  |
| %S | The SysOp's first name in mixed case.  |
| %t | The amount of time the user has left (in minutes).   |
| %T | The amount of time the user has left (in seconds).   |
| %u | The user's lastread pointer number. This token translates into a unique integer for each user on the system. This number is guaranteed not to change, even if the user file is sorted.                                     |
| %U | Translates to a simple underscore.   |
| %v | Upload path for the current file area (with trailing backslash).   |
| %V | Upload path for the current file area (with no trailing backslash).  |
| %w | The path to the current <b>files.bbs</b> -type file. This takes into account the alternate names which may be used by the <b>FileList</b> option.  |
| %W | The "steady baud rate," as passed via the -s command line parameter.   |
| %x | Drive letter of current drive, in upper case.  |
| %X | The last-read message number for the current message area.   |
| %Y | The user's current language number. (0 is the first language in language.ctl; 1 is second, and so on.)   |
| %Z | Translates to the user's full name, in caps.   |

---

In addition to the above translation characters, Maximus also supports a similar set of translation characters for display filenames:

In any **Display\_File** command or **.bbs** filename, Maximus can use any of the above external program translation characters. However, the first character of the sequence must be a “+” rather than a “%”.

For example, to display a file called **d:\#.bbs**, where # is the current node number, you can include the following command in the menu control file:

```
Display_File      D:\+k.bbs      Demoted      "Display it!"
```

Remember, the “+” is only used when specifying a translation character for a display filename. The percent sign is used in all other cases.

Lastly, one shortcut can also be used for menu names. If you wish to substitute the current node number in a menu filename, insert the “\*” character.

For example, the following line:

```
First Menu MAIN*
```

causes node 0 to display a menu called **main00.mnu**; node 1 will display **main01.mnu**, and so on. (The task number is in hexadecimal, so node 12 would display **main0c.mnu**.)

## 6.6. Running Doors

A *door* is just a fancy name for external programs that can communicate with an on-line user. Door programs contain routines to communicate with the modem, allowing the door to ensure that the user does not drop carrier and to check the user’s time limit.

However, some doors presents special problems. Several conflicting standards exist for *door interfaces*. The door interface describes how the calling BBS program (Maximus) interfaces with the door program. For example, most doors need to know the user’s name, whether or not the user supports ANSI graphics, and so on.

Maximus includes the capability to directly write any text-based door interface file using a simple MECCA file. (In addition, MEX can be used to write almost any binary-based door interface file.)

The distribution version of Maximus comes with MECCA files which allow you to create door interface files for the following formats:

- **dorinfo1.def** (QuickBBS and RBBS)

- **chain.txt** (WWIV)
- **callinfo.bbs** (WildCat!)
- **door.sys** (GAP and others)

In addition, you can write your own MECCA or MEX programs to generate almost any other type of text or binary-based door interface file.

This functionality is achieved through the *[write]* MECCA token. The *[open]* and *[post]* tokens are also used when writing door interface files.

The *[write]* token simply writes a line of text to a file opened with *[open]*, but it also makes translations to the string using external program translation characters.

For example, to instruct Maximus to write a QuickBBS or RBBS-compatible **dorinfo1.def** file, simply copy the following MECCA script into a file called **dorinfo.mec** and compile it. (The standard distribution version of Maximus includes this file in `\max\misc\dorinfo.mec`.)

```
[delete]Dorinfo1.Def
[open]Dorinfo1.Def
[write]%N[ comment      Write the BBS name          ]
[write]%S[ comment      Write the SysOp's first name   ]
[write]%s[ comment      Write the SysOp's last name    ]
[islocal write]COM0[ comment      Write the COM port    ]
[isremote write]COM%P[comment      (local is always COM0) ]
[write]%b BAUD,N,8,1[comment Write the baud rate      ]
[write] 0[ comment      Say that we're not networked   ]
[write]%A[ comment      Write the user's first name    ]
[write]%B[ comment      Write the user's last name     ]
[write]%c[ comment      Write the user's city          ]
[write]%g[ comment      Write the user's graphics      ]
[write]%o[ comment      Write the user's security level]
[write]%t[ comment      Write the user's time remaining]
[write]-1[ comment      Say that we're using a FOSSIL  ]
[quit      comment      And we're done!                ]
```

You can create similar files for other door interface types by simply creating another MECCA file with the appropriate commands.

Before running an external program, Maximus can create the **dorinfo1.def** file (or any of the above-mentioned files) in one of three ways:

*Create dorinfo1.def from a .mec file.* Simply include the following line before the call to *[xtern\_dos]* or *[xtern\_run]*:

```
[link]C:\Max\Misc\Dorinfo
```

As mentioned earlier, the distribution version of Maximus also comes with MECCA scripts to generate several other types of door interfaces. The format for using these interface files is similar:

```
[link]C:\Max\Misc\WWIV      - To create CHAIN.TXT
```

```
[link]C:\Max\Misc\CallInfo - To create CALLINFO.BBS
[link]C:\Max\Misc\DoorSys - To create DOOR.SYS
```

Create *dorinfo1.def* from a menu option. Similarly, you can achieve the same results through a menu option. Simply link the appropriate door interface **.bbs** file to the menu option containing the command to run. (For more information on linking menu commands, please see section 4.7.3.

For example, to instruct Maximus to create a **dorinfo1.def** file for a program called "C:\Max\Prg.Exe", use this in the menu control file:

```
Display_File C:\Max\Misc\Dorinfo Normal "RunPrg"
NoDsp Xtern_Run C:\Max\Prg.Exe Normal "R"
```

This concept can be applied to the other door interface types by substituting the name of the door script for "C:\Max\Misc\Dorinfo".

Create *dorinfo1.def* automatically for all menu programs. If you wish to have Maximus write **dorinfo1.def** every time it exits for an external program from a menu option, simply edit the **Uses Leaving** statement in the system control file:

```
Uses Leaving C:\Max\Misc\Dorinfo
```

This instructs Maximus to create **dorinfo1.def** whenever Maximus runs an external program from a menu option.

## 6.7. On-Line User Record Modification

Some door programs are written specifically for Maximus, and these programs occasionally need to directly change part of a user's profile, such as the user's remaining time, ANSI/AVATAR preference, phone number, and so on. Some of these programs need to do this even while the user is on-line.

Maximus supports on-line user record modification for most exit types. When instructed to do so, it will re-read the **lastus##.bbs** file for the current node after returning from an external program.

If you are running the external program as a menu option, then the fastest way to enable on-line modification is to place the **ReRead** modifier in front of the usual **Xtern\_\*** option. In other words, instead of invoking the program like this:

```
Xtern_Run D:\Path\Prog.Exe Demoted "Prog"
```

you must place add a **ReRead** modifier as follows:

```
ReRead Xtern_Run D:\Path\Prog.Exe Demoted "Prog"
```

Similarly, you can perform the same operation when using the *[xtern\_\*]* MECCA tokens by using an “@” as the first character of the program name. In other words, instead of invoking the program like this:

```
[xtern_run]D:\Path\Prog.Exe
```

you must use this instead:

```
[xtern_run]@D:\Path\Prog.Exe
```

However, keep in mind that most programs do not need this feature. For security reasons, you should not use this feature unless the external program's documentation states that on-line modification is required.

## 6.8. Doors and OS/2

OS/2-based communications programs can only run other OS/2-based programs as doors. Normally, DOS-based doors cannot be run under the OS/2 version of Maximus.

However, if you are using the third-party SIO.SYS communication driver, some DOS doors can be made to work with the OS/2 version of Maximus. See the documentation that comes with SIO for more information.

Aside from this restrictions, running OS/2-specific doors is usually much easier than running DOS-based doors. OS/2 programs do not have a 640k memory limitation, so there is no need for swapping or the **Xtern\_Erlvl** feature.

One feature of OS/2 communications programs is that they all use “port handles” when passing control from one program to another. A port handle is created by the DosOpen system call when an application opens a COM port. The port handle is different from the port number; the handle is assigned automatically by the operating system, and it is not the same as the port number.

To allow other applications to access the port, the program that “owns” the port spawns the door program directly, giving it the port handle number for the open communications port. The spawned program must use this handle to communicate with the port. (For example, when spawning Maximus from a front end mailer program, the “-p” command line parameter is used to pass a port handle number.)

Due to OS/2's file handle sharing architecture, any attempt to access the port without using the handle number will fail. This is why DOS doors cannot normally be used under Maximus-OS/2; DOS programs do not know about port handles, so they cannot inherit the port from an OS/2 communications program. However, see the SIO.SYS documentation for information on how to get around this problem.



---

## 7. Multinode Operations

Maximus supports an integrated paging and internode chat facility, making Maximus the ideal choice for multinode systems. Maximus is also LAN-friendly, meaning that it can run on any Netware, LANtastic, LAN Manager or LAN Server network, in addition to any other DOS-based network that supports file sharing.

To run a multinode version of Maximus, you need at least one of the following:

- MS-DOS/PC-DOS with a multitasking environment (Windows or DESQview),
- OS/2, Windows NT, Windows 95, or any other operating system with built-in multitasking capabilities, or
- two or more computers running networking software.

With the first two options, you can run multiple copies of Maximus on a single computer. With the latter option, you can run a single copy of Maximus on multiple networked computers. Combinations of the two approaches are also possible, such as networking two machines that each run four copies of Maximus.

If you decide to run multiple copies of Maximus on a single computer, you should ensure that your hardware is fast enough to run all nodes at full speed. The exact hardware requirements vary based on the system manufacturer, bus architecture, clock speed, available system memory, and the speed of the modems on your system.

However, some general rules are:

- If you are running the DOS version of Maximus, unless you have a very fast computer, it is unwise to run more than four nodes per machine. DOS was not designed to handle multitasking applications, and a lot of CPU time is wasted performing unnecessary context switching and polling for hardware events.
- If you are running the OS/2 version of Maximus, you can easily run 16 nodes (or more) on a fast computer, as long as you are using intelligent serial port hardware. The following intelligent serial boards are known to work with the OS/2 version of Maximus:

- ◆ IBM's ARCTIC (RIC) card
- ◆ DigiBoard's "DigiBoard/i" series
- Without an intelligent serial board, the OS/2 version of Maximus can probably handle 8 nodes on one machine, depending on your system's clock speed and other hardware capabilities.
- If you are not using an intelligent serial board, for either the DOS or OS/2 version of Maximus, you will require a 16550 UART (serial interface chip) on all serial ports that are serviced by Maximus.

Many high-speed internal modems come with 16550 chips built in, but if you are using external modems, many older serial port boards only have the inferior 8250 or 16450 chip. The 8250 and 16450 are usually socketed, so most computer dealers will be able to replace them for you by simply swapping in new chips.

The 16550 chip includes a number of buffering features that prevent characters from being lost during periods of high system load. The 16550 can also accept data much more quickly than the 16450.

## 7.1. Installation

Installation of a multinode version of Maximus is identical to the installation procedure for a single-node version of Maximus. However, you may want to keep these points in mind:

- If you are installing Maximus on a network, you should install it on a drive that can be accessed using the same drive letter on all workstations and servers that will be running copies of Maximus. If you are running Maximus on a non-dedicated server, you should probably run the Maximus session through the redirected version of the drive.

For example, assume that you have installed Maximus on `\\MYSERVER` on the `d:\max` directory. In addition, assume that all workstations access the Maximus drive with a "net use w: \\myserver\max" command.

Given this configuration, you should set up all of the Maximus control files to look for Maximus on the W: drive. (In fact, it is better to install Maximus directly to the W: drive and let the install program configure these settings for you.)

If you wish to run Maximus on the server machine, you should still execute a "net use w: \\myserver\max" and run the system from the W: drive, even though the information is stored locally in `d:\max`. (Otherwise, you would need

two separate sets of control files — one that pointed to **d:\max** and one that pointed to the W: drive.)

- Maximus-OS/2 supports UNC paths, so you can directly specify directories and filenames such as **\\myserver\max\max\misc\welcome.bbs**. However, many other programs do not accept UNC paths, so you should use this option with caution.
- You will normally need a separate batch or command file for each copy of Maximus that you wish to run. However, you only need one copy of the Maximus executables and system files.

When writing batch files to start Maximus, the command line switches shown below in Table 7.1 allow you to tailor Maximus to run as a different node number.

**Table 7.1 Multinode Command Line Switches**

| Parameter | Description                             |
|-----------|---|
| -p<num>   | Specify an alternate COM port.          |
| -b<speed> | Specify an alternate maximum baud rate. |
| -n<node>  | Specify an alternate node number.       |
| -l<file>  | Specify an alternate log filename.      |

The -n and -l parameters allow you to adjust the task number and log filenames at runtime. All nodes need a distinct node number, and all nodes need a separate log file.

In addition, if you are running multiple nodes of Maximus on the same machine, you will probably also need to configure each node to use a separate COM port number and speed, using the -p and -b parameters.

Please see Appendix C for more information on these command line parameters.

- Using DOS or OS/2 environment variables, you can actually run multiple copies of Maximus with only one batch file. If you set an environment variable to the current node number, like this:

```
set THISNODE=2
```

you can then refer to that variable within a batch file as **%THISNODE%**. Similarly, you can also set up variables for the port number, baud rate, log file, and so on.

You can then create a **runbbs.bat** or **runbbs.cmd** that contains a command like this:

```
max -p%thisport% -n%thisnode% -l%thislog%
```

As long as the environment variables are set up correctly before executing your **runbbs.bat**, you only need to maintain one copy of the file.

- If you wish to display node-specific information to users, you can use the “\*” token in the names of **.bbs** display files and system menus. The “\*” will be replaced with a two-digit hexadecimal task number. This allows you to add a command such as this:

```
Display_File    Misc\Bullet*    Demoted "Bulletins"
```

On node 1, selecting this option would display **\max\misc\bullet01.bbs**.

- All copies of Maximus must be started from the same directory. This allows you to share some files between nodes, in addition to providing a clean directory structure.
- If you are part of a FidoNet technology network, you may only want to run a front end mailer on one line. The internal WFC subsystem can be enabled on a node-by-node basis: simply include a **-w** on the command line for those nodes that are to run in WFC mode.
- In your system startup file (either **autoexec.bat** for DOS or **startup.cmd** for OS/2), you should include commands to delete the following files from the Maximus directory structure:

```
\max\active*.bbs
\max\qwk_busy.$$$
\max\ipc\*.bbs
```

These temporary files are created during normal Maximus operation. However, if a Maximus session ends abnormally, some of the files above may be left around. These files should be deleted to prevent confusion when Maximus starts up again.

In the case of a network installation, the commands to delete the above files should be placed in the startup file on the server, not on the workstations.

- If you wish to use the multinode chat or the paging features, your operating system must support file and record locking. Under DOS, this means that you must load the **share.exe** program, as described in the installation instructions. (Under OS/2 and Windows 95, file locking is built into the operating system, so no special utilities are necessary.)
- Ensure that all copies of Maximus have a unique and *non-zero* node number. If the task number is set to zero, Maximus will assume that your system is run-

ning in a single-node environment, so that node will not be able to communicate with the rest of the system.

However, if you wish to create a “hidden” node for local logons, you may want to configure that node to run as node 0. This node will be invisible to the rest of the system, but it will otherwise function normally.

- For information on installing Maximus on a networked OS/2 system, please see the Master Control Program guide in section 7.3.

## 7.2. Multinode Chat Operation

Maximus includes a built-in multinode chat and paging facility. Users can be paged by others, participate in real-time conferences (both public and private), display a list of on-line users, and more.

**DOS only!** For DOS users, the first step in configuring the multinode chat is to enable the **Path IPC** statement in the system control file. For optimal performance, this directory should be placed on a RAM disk, but it can also be placed in a normal directory.

For either OS/2 or DOS users, the next step is to edit the menus control file and ensure that the **Display\_Menu CHAT** option is uncommented.

Having made these changes, recompile the control files and log on locally to test the system. (You will need to use two different user names, since Maximus only allows a user to log onto one node at a time.)

Before testing chat mode, enter the Chat Section and look at the menu display. The table should show the list of on-line callers. If the display is blank, something is not configured correctly:

- Under DOS, ensure that you have loaded **share.exe**, as indicated in the installation instructions.
- Under DOS, ensure that the **Path IPC** for all nodes points to the same directory.
- Under OS/2, ensure that the **mcp.exe** program is in the main **\max** directory or somewhere else on your **PATH** statement.

If the menu display seems to be in order, try toggling your “chat availability” flag a few times. After your status has been toggled, the “Status” portion of the table should indicate whether or not you are available for chat. Then switch to the other node and redisplay the chat menu. You should see that the status of the first node is also reflected on the screen of the second node.

Finally, after confirming that everything else is working properly, you can enter multinode chat. To initiate a chat, select the **Page** option and enter the number of the node to be paged. Under DOS, it may take up to 15 seconds for the chat request to register on the other node, but under OS/2, the other user should be notified instantly.

On the other user's console, you should see a "You are being paged by John Doe (node ##)" message. This is the standard paging message; to modify it, you can either edit the **english.mad** language file, or you can create a separate display file as `\max\misc\chatpage.bbs`.

To answer the chat request on the other node, select the **Answer Page** option and enter the node number of the user who sent the request. This will place the user in chat mode as well. The first user should see a "Jane Doe joins the conversation" message, which indicates that the other user answered the chat request.

The user who answered the page will not see anything immediately; to find out who is participating in the conversation, simply type a "/w" command at the beginning of a line. To list all of the callers on the system, whether or not they are in chat mode, type "/s".

Once in chat, users can send messages to each other by simply typing the text that they wish to send. Maximus will automatically word-wrap at the end of lines, and the text will be transmitted one line at a time. Try typing a few lines from each node to ensure that the chat function is working properly.

Once you are finished testing, you can use the "/q" command on each node to exit chat mode. (When a node exits chat, the other nodes participating in the same chat should see a "John Doe leaves the conversation" message.)

In addition to the private chat facility, Maximus also supports a group chat, or a "virtual CB channel." The CB chat is useful when you have three or more nodes and want to have more than two callers participating in a conversation. Maximus supports up to 255 concurrent "channels," which means that there can be up to 255 separate conversations going on at the same time.

However, the CB chat has no paging ability; it is up to the callers to look at the status screen in the Chat Section and find out which channel is being used by the other users.

For more information on using Maximus's multinode chat, please see the chat help file. (To display the help file, enter "/" from inside chat mode.)

### 7.3. Master Control Program

**OS/2 only!** The OS/2 version of Maximus uses the Master Control Program server (MCP) for all multinode communication. When Maximus loads, it automatically starts the MCP server and runs it as a background task. The server then creates a number of named pipes that are used for communication among Maximus nodes.

On a non-networked machine, this is ideal, since all Maximus tasks reside on the same machine, and they all communicate with the local MCP server by default.

However, if you are running copies of Maximus on multiple workstations, they will try to talk to the MCP running on the workstation, not on the server. To prevent this, you must start a copy of MCP on the server yourself, and you must tell each node where to find the MCP server.

*Starting the MCP server.* The MCP server should be started from the **config.sys** file on the network server, as shown below:

```
RUN=c:\max\mcp.exe . \pipe\maximus\mcp <nodes> server
```

The **nodes** parameter tells MCP to support up to **nodes** concurrent tasks. This parameter should have the same value as the **MCP Sessions** definition in the system control file. The **nodes** parameter should be greater than the maximum number of Maximus nodes that you expect to run at one time. (It is an error to specify too few nodes, but you can specify more nodes without worry.)

*Configuring the workstations to use the MCP server.* In the main system control file, the **MCP Pipe** definition can be used to configure the location of the MCP server. The default pipe is **\pipe\maximus\mcp**, but you must change this to point to the server.

If MCP is running on the server called \\myserver, you must edit the **MCP Pipe** definition to read:

```
MCP Pipe    \\myserver\pipe\maximus\mcp
```

In addition, you can also use the **-a** command line parameter to override the **MCP Pipe** setting in the system control file.





---

# 8. Utility Documentation

This section describes the command line interface of the various utility programs that are included in the Maximus distribution.

## 8.1. ACCEM: MECCA Decompiler

### 8.1.1. Description

ACCEM is the inverse of the MECCA utility: ACCEM reads a compiled **.bbs** file and converts it into a human-readable **.mec** file. This functionality can be useful if you have lost the source for one of your **.bbs** display files, or if you are trying to change a compiled **.bbs** file which was given to you by someone else.

After running ACCEM on a **.bbs** file, you can freely edit the resulting **.mec** file and recompile it as you wish. The **.mec** file created with ACCEM should be identical to the original **.mec** file, with one small exception: label names are not stored in the **.bbs** file, so ACCEM will create numbered labels. For example, the following MECCA sequence:

```
[goto foo]

[/foo]
```

could be decompiled into the following:

```
[goto L1]

[/L1]
```

Although the labels are different, the **.mec** file will still compile correctly.

### 8.1.2. Command Line Format

The command line format for ACCEM is:

```
accem infile [outfile] [-s]
```

**infile** is the name of the **.bbs** file to convert. If no extension is given, ACCEM assumes an extension of **.bbs**.

**outfile** is the optional name of the **.mec** output file. If this parameter is omitted, ACCEM will assume the name specified for **infile**, but using a **.mec** extension.

The **-s** switch instructs ACCEM to split lines which are longer than 100 characters. When ACCEM needs to split a line, it will place an opening brace in the 100th column of the line, and it will place the closing brace at the beginning of the next line. This option is useful if your text editor can only display a limited number of columns.

For example, to convert **test.bbs** to **test.mec**, any of the following commands will work:

```
accem test
accem test.bbs
accem test.bbs test.mec
```

To split lines which are over 100 characters in length, the following commands would also work:

```
accem test -s
accem test.bbs -s
accem test.bbs test.mec -s
```

## 8.2. ANS2BBS/MEC: ANSI to MEC conversion

### 8.2.1. Description

ANS2BBS and ANS2MEC convert ANSI graphics into MECCA tokens. ANS2BBS and ANS2MEC can convert almost any file containing ANSI graphics into an equivalent MECCA display file that can be processed by Maximus.

ANS2BBS will convert a file containing ANSI graphics directly into a **.bbs** file that can be displayed by Maximus. ANS2MEC converts an ANSI file into a MECCA source file. The created **.mec** file can then be edited and compiled as usual.

ANS2BBS is useful for a one-time translation, but ANS2MEC is best if you wish to add some special effects to the display file or clean up some of the display codes.

### 8.2.2. Command Line Format

The format for ANS2BBS (and ANS2MEC) is as follows:

```
ans2bbs infile [outfile]  
ans2mec infile [outfile]
```

**infile** is the name of the input ANSI graphics file. If no extension is provided, an **.ans** extension is assumed by default.

**outfile** is the optional name of the output file. For ANS2BBS, the default output file has a **.bbs** extension, whereas for ANS2MEC, the default output file has a **.mec** extension.

ANS2BBS and ANS2MEC will try to do the best job that they can when converting an ANSI file, but due to some ambiguities in the ANSI cursor-movement syntax, they cannot always correctly convert all ANSI graphics files. ANS2BBS and ANS2MEC have problems with some “highly-animated” screens, particularly those generated by the “Diagonal,” “Gate,” “Squiggle,” and other fancy drawing modes of the TheDraw program by Ian Davis. However, ANS2BBS and ANS2MEC can handle almost all straight-through ANSI files, so unless you are using one of those scanning modes, you should not have any problems.

Once you have converted an ANSI screen, it is best to put it in a place where you can test it in local mode (or display it using the ORACLE utility). If the file did not convert correctly and has formatting glitches, you have three choices:

- If the file is animated, load the file using TheDraw, turn off animation mode by pressing Alt-J and then *N*. Save the file and try ANS2BBS/ANS2MEC again.
- Convert the file using ANS2MEC, and edit the resulting MECCA file to correct the problem.
- Leave the ANSI file as-is and display the ANSI version of the file directly to callers. Although the ANSI codes will not display properly on the local screen, they should be displayed normally for remote callers.

## 8.3. CVTUSR: User File Conversions

### 8.3.1. Description

CVTUSR converts foreign user files into the Maximus 3.0 user file format. CVTUSR can handle user files in the Maximus 1.0, Maximus 2.0, QuickBBS, RA 2.00, and Opus 1.0x formats.

### 8.3.2. Command Line Format

The command line format for CVTUSR is:

CVTUSR **params**

The **params** parameter can be one of the command line parameters shown in Table 8.1 below:

Table 8.1 CVTUSR Command Line Switches

| Switch | Description  |
|--------|--|
| -l     | Reset the lastread pointers in a Maximus 3.0 user file. This option is normally only used to fix cross-linked lastread pointers. Using this function when it was not explicitly requested (by a note in the Maximus log file) may destroy the lastread pointers for existing users.  |
| -n     | Convert an old Maximus version 1.x user file to the Maximus 3.0 format.  |
| -o     | Convert an Opus 1.10-style <b>user.dat</b> file to Maximus 3.0 format.<br><br>This procedure converts almost all of the Opus 1.10 user file fields, with the exception of the expiry dates, personal welcome screens, and any utility-specific fields which may be stored in the user file.  |
| -p     | Convert a Maximus version 2.x user file to the Maximus 3.0 format.   |
| -q     | This switch tells CVTUSR to convert a QuickBBS or RA 2.00 <b>users.bbs</b> to a Maximus 3.0 user file. This conversion is not as complete as some of the others; for example, it will not convert the ANSI graphics and “More” prompt settings.<br><br><b>WARNING!</b> If your version of RA encrypts the passwords in the RA user file, CVTUSR will not be able to convert the user file. |
| -s     | This flag tells CVTUSR to swap the “alias” and “name” fields in the current Maximus 3.0 user file.   |

## 8.4. EDITCAL: Call Modification Utility

### 8.4.1. Description

EDITCAL modifies the “number of callers to system” count in the **bbstat##.bbs** files. This program is useful if you have recently changed from another BBS pack-

age, but you want to set the caller count to reflect the actual number of callers to your system.

#### 8.4.2. Command Line Format

The command line format for EDITCAL is:

```
editcal node [num_calls]
```

**node** is the node number whose “number of callers” count is to be modified.

**num\_calls** is the new “number of callers” value for the specified node. If this parameter is omitted, EDITCAL will display the current “number of callers” count for the node.

## 8.5. FB: File Database Compiler

#### 8.5.1. Description

FB is the Maximus File Database compiler. FB compiles the ASCII listings in the **files.bbs** directory listings into a format which can be used by the global downloading routines, the upload duplicate file checker, and the Fast Locate feature. Maximus can use **files.bbs** directly in most situations, but many file area functions will run much faster if you use FB.

By default, FB will extract information about file sizes and dates from the directory entries in the area’s download directory. However, if you have enabled **Type Date-List** in a file area definition, FB can also extract file size and date information from the ASCII file listing.

#### OS/2 only!

When you have file areas stored on an HPFS drive, FB will automatically use the HPFS “creation date” as the file upload date, and it will use the HPFS “last write date” as the file’s true date. When requested to perform a new files search, Maximus will compare the requested date with the file’s upload date. However, when displaying the file catalog, Maximus will show the file’s true date.

#### 8.5.2. Command Line Format

The command line format for FB is:

```
fb switches [area ...]
```

The *switches* parameter must be one or more of the switches in Table 8.2 below:

**Table 8.2 FB Command Line Switches**

| Switch   | Description  |
|----------|--|
| -a       | Compile all file areas.  |
| -f<file> | Use <file> instead of the default farea.dat file area database.  |
| -p<file> | Use <file> as the Maximus .prm file. This switch overrides the <b>MAXIMUS</b> environment variable. FB uses the .prm file to obtain information about the main Maximus system directory, the name of the file area database, and other file area settings. |
| -r       | This switch forces FB to read information directly from the file directory, rather than reading from the file area list. This switch only affects areas declared using the <b>Type FileList</b> keyword.   |
| -s       | Skip areas marked as “Type Slow” or “Type CD.”   |
| -u       | Instead of processing the download paths for the requested file areas, process the upload paths instead. This parameter is used internally by the <b>runfb.bat</b> and <b>runfb.cmd</b> files. (See below for more information.)                           |
| -x       | Do not perform the final merge to <b>maxfiles.idx</b> .  |

[**area ...**] is the optional list of areas to process. If a list of file areas is provided, FB will only include the listed areas in the file database. The wildcard “\*” may be used at the end of a name to match any number of characters. (To build all file areas, use the “-a” switch instead.)

Examples:

```
fb -a
```

This command builds the file database for all file areas.

```
fb -s -a
```

This command builds the file database for all file areas, except those on CD-ROM.

```
fb 1 2 3 4
```

Process only the files in areas 1, 2, 3 and 4.

```
fb os2.* dos.*
```

Process all file areas that start with “os2.” or “dos.”

### 8.5.3. Database Files

When compiling a file area, FB will parse **files.bbs**, and for each file area, it will create the files shown in Table 8.3:

**Table 8.3 FB Database Files**

| Filename  | Description   |
|-----------|---|
| files.dat | A compiled version of each file's name, size, timestamp, privilege level and flags. |
| files.dmp | A compiled version of each file's description.                                      |
| files.idx | A sorted binary index of all files in the current area.                             |

If you are using the **FileList** keyword in the file area definition, Maximus will remove the extension of the **FileList** file and add **.dat**, **.dmp** and **.idx** as appropriate. For example, if you specified the following in a file area definition:

```
FileList D:\Area1.Lst
```

FB would create files called **d:\area1.dat**, **d:\area1.dmp** and **d:\area1.idx**. This allows owners of CD-ROMs to store all of the file area information in an alternate location.

### 8.5.4. Database Building for Uploads

Normally, after a user has uploaded a batch of files, the file database needs to be updated with the name and location of the uploaded files. Once the user has entered all of the file descriptions, Maximus tries to find a file called **\max\runfb.bat** or **\max\runfb.cmd**. If found, Maximus will execute it with the following parameters:

```
runfb farea_dat areanum -u
```

The **farea\_dat** parameter is the name of the file area data file.

The **areanum** parameter is the name of the current file area.

The **-u** parameter indicates that FB should build the database based on the upload paths.

The standard **runfb.bat** file simply calls FB with the same parameters as passed to it. Unfortunately, the database build process can take a long time for systems that have many files.

In the default distribution, **runfb.bat** looks like this:

```
fb %1 %2 %3
```

However, this line can be modified so that the file database is updated after the user logs off:

**DOS only!**      `echo fb %1 %2 %3 >>do_fb.bat`

**OS/2 only!**      `echo fbp %1 %2 %3 >>do_fb.cmd`

The above command creates a log of file areas to be updated. Your **runbbs.bat** should execute the following command after processing each caller:

**DOS only!**      `if exist do_fb.bat call do_fb.bat  
if exist do_fb.bat del do_fb.bat`

**OS/2 only!**      `if exist do_fb.cmd call do_fb.cmd  
if exist do_fb.cmd del do_fb.cmd`

These lines cause Maximus to perform all file database updating after the caller logs off, which saves on both memory and on-line time. Ensure that the above commands are run after *every* caller, regardless of whether or not the caller entered NetMail, EchoMail, or no mail.

## 8.6. MAID: Language File Compiler

### 8.6.1. Description

MAID is the Maximus Language File compiler. MAID takes a language definition, such as **english.mad**, and turns it into a form usable by Maximus. The language file can be used to support non-English languages, or it can be used to simply change the prompts in the English version of Maximus.

### 8.6.2. Command Line Format

The command line format for MAID is given below:

```
MAID langname [switches]
```

The **langname** parameter is the full path and name of the language file. Do not include the **.mad** extension.



The optional **switches** parameter can be zero or more of the switches shown in Table 8.4 below:

**Table 8.4 MAID Command Line Switches**

| Switch   | Description  |
|----------|--|
| -d       | Generate the dynamic language include file. The <b>langname.h</b> file is not created unless you use this switch.  |
| -n<name> | Use the name < <b>name</b> > as the name of the language. This is the name that is displayed on the <b>Chg_Language</b> menu when the user is asked to select a language.<br>By default, MAID uses the base filename of the language file as the language name. However, the -n parameter can be used to specify an explicit language name. To include spaces in the language name, enclose the entire parameter in quotes like this:<br>maid english "-nAmericanEnglish"      |
| -p<prm>  | After compiling a language file, MAID will automatically update the Maximus parameter file specified by the <b>MAXIMUS</b> environment variable. However, if you want MAID to update a different <b>.prm</b> file, the -p switch can be used to override the <b>MAXIMUS</b> environment variable.<br>Without this switch, and without a correctly-set <b>MAXIMUS</b> environment variable, the system control file must be recompiled every time you change the language file. |
| -s       | Generate the static language include file. The <b>langname.lth</b> file is not created unless you use this switch.   |

### 8.6.3. Language-Related Files

MAID reads language source information from a file called **langname.mad**. The distribution version of Maximus comes with one language file called **english.mad**.

The different input and output files used by MAID are described in Table 8.5:

**Table 8.5 MAID Input and Output Files**

| Extension | Description   |
|-----------|---|
| .mad      | The <b>Maximus International Definitions</b> file. This file contains the “source” for the language and is the input to MAID. This file can be edited with an ordinary text editor. |
| .ltf      | The <b>Language Translation File</b> . This is the compiled version of the <b>.mad</b> source file.   |
| .lth      | The dynamic language include file for C programs.   |
| .h        | The static language include file for C programs.  |
| .mh       | The dynamic language include file for MEX programs.   |

For information on modifying the system language files, please see section 18.12.

## 8.7. MAXPIPE: OS/2 Redirection Utility

### 8.7.1. Description

MAXPIPE is an OS/2-only program used to redirect the I/O of command line programs to the COM port. This program is useful when spawning an OS/2 shell, or when running certain programs that only use console I/O. (For example, Maximus-OS/2 automatically calls MAXPIPE when spawning archivers to compress QWK packets.)

MAXPIPE also provides a “watchdog” facility. If the user drops carrier while the external program is active, MAXPIPE kills the running process and returns to Maximus.

### 8.7.2. Command Line Format

The command line format for MAXPIPE is:

```
MAXPIPE handle program [args ...]
```

**handle** is the COM port handle, as generated by the “%P” external program translation character. If a handle of “0” is used, MAXPIPE will run in local mode.

**program** is the name of the external program to be spawned. Ensure that the full filename and path is provided.

[**args** ...] are the optional arguments to pass to the external program.

|           |
|-----------|
| Title: Af |
| Creator:  |

MAXPIPE works only with programs that use “stdin/stdout” output. Programs which write directly to the console (or programs which use Presentation Manager output calls) will not function correctly with MAXPIPE.

## 8.8. MECCA: Display File Compiler

### 8.8.1. Description

MECCA compiles **.mec** source files into binary **.bbs** files that can be displayed by Maximus. MECCA source files can contain human-readable tokens to change the text color, display simplistic menus and prompts, and other display-oriented tasks.

### 8.8.2. Command Line Format

The command line format for MECCA is:

```
MECCA infile [outfile] [-t] [-r]
```

**infile** is the name of the input file. If no extension is specified, MECCA assumes **.mec** by default. **infile** can also include wildcards.

**outfile** is the optional output filename for the compiled MECCA file. This parameter is optional, and if not specified, it defaults to **infile.bbs**.

The optional **-t** parameter instructs MECCA to compare the date stamps of the input and output files, and if the output file is newer than the input file, to skip compiling that file. This is useful for recompiling an entire directory of **.mec** files.

The optional **-r** parameter instructs MECCA to produce a **.rbs** file (instead of a **.bbs** file) for *RIPscrip* graphics support. In addition, this switch disables RLE compression, since RLE can sometimes interfere with *RIPscrip* graphics sequences.

Please see section 17 for information on the format of MECCA files.

## 8.9. MR: Maximus Renumbering Program

### 8.9.1. Description

MR is a \*.MSG format message renumbering program. MR is not required if your system uses only Squish-format areas.

MR automatically reads the information given in the message area data file, and it then renumbers, deletes and relinks messages in \*.MSG-style areas.

Renumbering is useful for eliminating large gaps in the message numbers of \*.MSG areas (which are created when users delete messages). MR can also purge messages

based on message age or the total number of messages in the area, allowing you to maintain a roughly-constant size for your message bases.

### 8.9.2. Command Line Format

The command line format for MR is:

```
mr options area [area ...]
```

The **options** parameter can be zero or more of the command line switches shown in Table 8.6:

**Table 8.6 MR Command Line Switches**

| Switch   | Description   |
|----------|---|
| -p<file> | Use the <b>.prm</b> file specified in <file> instead of the setting in the <b>MAXIMUS</b> environment variable. |
| -m<file> | Use <file> as the message area data file, rather than using the data file specified in the <b>.prm</b> file.    |

The **area** parameter must include one or more area names to be renumbered. To renumber all areas on the system, specify an area of “all”.

For example, to renumber all message areas:

```
mr all
```

To renumber message areas “muffin,” “tub” and “local”:

```
mr muffin tub local
```

### 8.9.3. Renumbering Operation

When renumbering, MR will examine the **Renum Days** and **Renum Max** settings for each \*.MSG message area. If either of those two keywords are set, MR will purge messages from the area based on the specified criteria. Messages can be killed by message number, by age, or both.

MR automatically updates the Maximus lastread files and message area links. To maintain \*.MSG areas, simply include a call to MR in your daily event batch file, and all of your renumbering and purging needs will be taken care of automatically.

## 8.10. ORACLE: Display File Viewer

### 8.10.1. Description

ORACLE is an off-line **.bbs** file viewer that allows you to view compiled **.bbs** files without logging on. ORACLE is a quick way to test changes made to standard **.mec** files.

### 8.10.2. Command Line Format

The command line format for ORACLE is:

```
ORACLE bbsfile [options ...]
```

**bbsfile** is the name of the compiled **.bbs** file that you wish to view. If no extension is supplied, **.bbs** is assumed.

The **options** parameter specifies zero or more of the switches from Table 8.7:

**Table 8.7 ORACLE Command Line Switches**

| Switch | Description   |
|--------|---|
| -hX    | Sets the current help level to X, where X is the first letter of a valid help level. (N = Novice; R = Regular; E = Expert.)   |
| -i     | Disables high-bit IBM characters. When this option is enabled, ORACLE will automatically translate IBM Extended ASCII to the standard ASCII equivalent.   |
| -kX    | Sets the user's keys to X, where X is a simple listing of keys to assign to the user. Valid keys are from 1 to 8 and A to X. For example, using "k1237AD" would give the user keys 1, 2, 3, 7, A and D.               |
| -mX    | Sets the local video mode to X, where X is a valid video mode. (B = BIOS; I = IBM.) This only applies to the DOS version of Maximus.  |
| -pX    | Reads the Maximus <b>.prm</b> information from the file X. This setting will override the <b>MAXIMUS</b> environment variable.  |
| -q     | This option enables hotkeys mode.   |
| -slX   | This option sets the virtual screen length to X rows. This does not change your physical screen length; however, it does tell Maximus when to display "More [Y,n,=]?" prompts. The default screen length is 24 lines. |
| -swX   | This option sets the virtual screen width to X columns. This does not change your physical screen width; however, it controls when virtual screen wraps occur.  |
| -t     | The -t parameter forces ORACLE into TTY video mode. This disables all ANSI and AVATAR graphics commands, and ORACLE displays files just as they would be shown to a TTY caller.                                       |

|     |  |
|-----|--|
| -vX | This sets the user's privilege level to X, where X is either a numeric privilege level or a user class abbreviation. |
|-----|--|

---

Settings from the ORACLE command line can also be set permanently using an environment variable. By issuing a “SET ORACLE=” command, you can create a set of defaults for every file that you view with ORACLE:

For example, issuing the following sequence of commands:

```
SET ORACLE=-v100 -q
ORACLE D:\Max\Misc\Bulletin
```

is identical to entering all of this at once:

```
ORACLE D:\Max\Misc\Bulletin -v100 -q
```

Although the first example looks like more typing, you can easily place the SET command into your **autoexec.bat** (DOS) or **config.sys** (OS/2), and then only type “oracle <filename>” whenever you want to display a file.

## 8.11. SCANBLD: \*.MSG Database Builder

SCANBLD builds databases of \*.MSG-format message areas. SCANBLD is not required if your system uses only Squish-format areas.

The primary function of SCANBLD is to speed up the internal Maximus mail-checker and **Msg\_Browse** commands. Accessing messages in \*.MSG areas is very slow, so SCANBLD builds an index of the messages in each message area to decrease processing time.

SCANBLD must be run after certain events occur, including after running a message renumbering utility, after receiving EchoMail, after a user enters a message, and so on. Running these commands is somewhat inconvenient, but SCANBLD is required for SysOps who still insist on running \*.MSG-format message areas.

### 8.11.1. Command Line Format

The command line format for SCANBLD is:

```
SCANBLD [switches...] [area...] [!area...]
        [All | Local | Matrix | Echo | Conf |
        @tosslog]
```

The optional **switches** parameter can specify any of the command line switches from Table 8.8 below:

**Table 8.8 SCANBLD Command Line Switches**

| Switch   | Description   |
|----------|---|
| -c       | Instructs SCANBLD to do a full compile of each area processed. By default, SCANBLD will try to update the mail database in the areas processed, without necessarily rebuilding the entire area. The -c switch should always be used after renumbering a message area. |
| -m<file> | Use the message area data file specified by <b>file</b> , rather than using the default data file specified in the <b>.prm</b> file.  |
| -nd      | Instructs SCANBLD to <i>not</i> delete the @tosslog filename after processing the entries within. This is useful if you have other utilities which need the tosslog after SCANBLD has finished.   |
| -p<file> | Use the <b>.prm</b> file specified by <b>file</b> , rather than using the setting in the <b>MAXIMUS</b> environment variable.   |
| -q       | Quiet mode. Instead of displaying the statistics for each area, display a single hash sign (“#”) instead.   |
| -u<file> | Use the user file specified by <file>, rather than the default user file specified in the <b>.prm</b> file.   |

The remainder of the SCANBLD command line contains a list of areas (or types of areas) to process:

If you specify **All**, SCANBLD will rebuild the database for all message areas.

If you specify **Local**, SCANBLD will rebuild the database for local message areas only.

If you specify **Matrix**, SCANBLD will rebuild the database for NetMail areas only.

If you specify **Echo**, SCANBLD will rebuild the database for EchoMail areas only.

If you specify **Conf**, SCANBLD will rebuild the database for Conference areas only.

If you specify **@tosslog**, SCANBLD will read the file specified by **tosslog** and read a list of area tags from within. SCANBLD will rebuild message areas which have a **Tag** keyword that matches one of the tags specified in the **tosslog** file.

If you specify the name of an area on the command line, SCANBLD will rebuild the message database for that area.

If you specify **!area** on the command line, SCANBLD will explicitly skip the specified message area, even if it was included by another command line parameter. (This option is useful in conjunction with the “All,” “Local,” “Matrix,” and other related keywords.)

### 8.11.2. Examples

The options specified on the SCANBLD command line are cumulative, so entering the following:

```
scanbld echo matrix 45 !22 @et.log
```

causes SCANBLD to process all EchoMail and NetMail areas (except for area 22), in addition to area number 45 and the areas listed in **et.log**.

To ensure that the mail database is always synchronized with your message base, you should run SCANBLD as follows:

After a user enters EchoMail (usually errorlevel 12):

```
SCANBLD local matrix @et.log
```

After a user enters NetMail (usually errorlevel 11):

```
SCANBLD local matrix
```

After a user enters local mail (usually errorlevel 5):

```
SCANBLD local
```

After importing EchoMail:

```
SCANBLD local matrix @et.log
```

After running any message-renumbering utility:

```
SCANBLD all /c
```

Finally, if you use an external message editor to access \*.MSG areas, you must run SCANBLD over all areas that were modified by the editor. If your editor produces an **echotoss.log**-like file, run SCANBLD after your editor using the command shown for “After a user enters EchoMail.”

However, if your external editor does not produce an **echotoss.log** (or similar) file, you must scan all areas using the following command:

```
SCANBLD all
```

If these instructions are not followed to the letter, SCANBLD may miss messages for your users which would be otherwise be flagged as new mail.



## 8.12. SILT: Control File Compiler

### 8.12.1. Description

SILT is the Maximus control file compiler. SILT compiles the raw ASCII control files into the binary parameter files that are used by Maximus. You must run SILT every time you make a change to one of the Maximus control files.

### 8.12.2. Command Line Format

The SILT command line format is:

```
SILT ctl_file [switches]
```

**ctl\_file** is the name of the control file to be compiled (without an extension). By default, if no parameters other than **ctl\_file** are specified, Maximus will compile all features in the control file. (However, it will not generate the Maximus 2.x-compatible **area.dat** by default.)

The optional **switches** parameter can specify zero or more of the command line switches from Table 8.9:

**Table 8.9 SILT Command Line Switches**

| Switch | Description  |
|--------|--|
| -2a    | Create a Maximus 2.x-compatible <b>area.dat</b> file. Area names are truncated to 10 characters.   |
| -2u    | Create a Maximus 2.x-compatible <b>area.dat</b> file. Area names are truncated to 10 characters, but dots (".") are converted to underscores ("_").                    |
| -2s    | Create a Maximus 2.x-compatible <b>area.dat</b> file. Only the last part of the area name (after the last dot) is written for message and file areas within divisions. |
| -a     | Compile message and file areas only.   |
| -am    | Compile only message areas.  |
| -af    | Compile only file areas.   |
| -m     | Compile only menu definitions.   |
| -p     | Compile only the system control files (and the other control files required for <b>max.prm</b> ).  |
| -u     | Run in "unattended mode." SILT will automatically create directories that do not exist, and it will not pause for user input.  |
| -x     | Compile everything (default).  |

## 8.13. SM: Session Monitor

### 8.13.1. Description

#### OS/2 only!

Session Monitor (SM) is a Presentation Manager program for Maximus-OS/2 SysOps. SM works in conjunction with MCP to allow SysOps to view and manipulate remote Maximus sessions, either on the local machine or across a LAN.

### 8.13.2. Command Line Format

SM has the following command line format:

```
sm
```

No command line parameters are supported.

### 8.13.3. Using SM

When SM is started, it will read information about your system from the **.prm** file specified by the **MAXIMUS** environment variable. It will then attempt to connect to the MCP server specified by the **MCP Pipe** definition in the system control file. SM can connect to a MCP server on a local machine, and if the **MCP Pipe** definition points to a network server, SM can also connect to a MCP server running on another machine.

Once connected to the MCP server, SM displays a system overview. For each node, SM will display:

- The name of the user on that node, or “(off-line)” if Maximus is not running.
- The status of the user (such as “Transferring a file” or “Available for chat”).
- The last “ping” time. All Maximus nodes will send a “ping” to the MCP server at a predefined interval. If the MCP server does not receive a ping from a Maximus node for a certain period of time, it will highlight the ping time in red. This indicates that there may be trouble on the indicated node.

To interact with a Maximus node, select the node with the mouse and press mouse button 2. If a user is logged on, a pop-up menu will present the following options:

- *View*. This option allows you to view the screen for the selected node. The screen may not be displayed properly if the node is in WFC mode or if the caller is running an external program.

## 8. Utility Documentation 139

- *Message*. This option allows you to send a message to the user logged onto the node.
- *Global>Message*. This option allows you to send a message to all active Maximus nodes.



---

# 9. REXX User File Interface

## 9.1. Introduction

**OS/2 only!** Maximus-OS/2 includes a REXX User File Application Programming Interface (API). This API allows programs written in REXX to scan the user file, read user information for particular users, update fields in existing user records, and add new users.

To use the REXX API, the **maxuapi.dll** file must be on your system's **LIBPATH**, and the following prologue must be inserted at the beginning of the REXX program:

```
/* rexx */  
  
call RxFuncAdd 'MaxLoadFuncs', 'MAXUAPI', 'MaxLoadFuncs'  
call MaxLoadFuncs
```

This code instructs the REXX interpreter to load a copy of **maxuapi.dll** and to import all of the REXX user file API functions.

Table 9.1 lists the functions supported by the REXX user file API:

**Table 9.1 REXX User API Functions**

| Function               | Description  |
|------------------------|--|
| MaxLoadFuncs           | Load all of the user file API functions into the REXX namespace.   |
| MaxUnloadFuncs         | Unload all of the user file API functions  |
| UserFileClose          | Close the user file. This function must be called when a REXX program is finished accessing the user file. |
| UserFileCreateRecord   | Create a new record in the user file.  |
| UserFileFind           | Find a single user within the user file.   |
| UserFileFindClose      | Terminate a multi-user find session.   |
| UserFileFindNext       | Find the next user in a multi-user find session.   |
| UserFileFindOpen       | Begin a multi-user find session.   |
| UserFileFindPrior      | Find the previous user in a multi-user find session.   |
| UserFileGetNewLastread | Obtain a lastread pointer for a new user.  |
| UserFileInitRecord     | Initialize the "usr." stem variable.   |

|                |  |
|----------------|--|
| UserFileOpen   | Open a user file.  |
| UserFileSize   | Retrieves the number of records stored in the user file. |
| UserFileUpdate | Updates (commits) changes to an existing user record.    |

For examples of using the REXX user file API, please see the sample **\*.cmd** files that are distributed with Maximus.

## 9.2. Function Descriptions

This section describes all of the functions supported in the REXX user file API.

---

### MaxLoadFuncs

---

#### Prototype

call MaxLoadFuncs

#### Description

The **MaxLoadFuncs** procedure loads all of the user file API functions into memory. This procedure must be called before any of the other API functions can be used.

Note that the **MaxLoadFuncs** procedure must also be loaded into memory before it can be called. To load the **MaxLoadFuncs** procedure (followed by the rest of the user file API functions), use the following REXX code:

```
/* rexx */

call RxFuncAdd 'MaxLoadFuncs', 'MAXUAPI', 'MaxLoadFuncs'
call MaxLoadFuncs
```

---

### MaxUnloadFuncs

---

#### Prototype

call MaxUnloadFuncs

#### Description

This procedure unloads all of the user file API functions from memory. **MaxUnloadFuncs** should be called at the end of a command file that contains calls to the user file API functions.

---

## UserFileClose

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | rc=UserFileClose( <b>huf</b> )  |
| <b>Arguments</b>    | <b>huf</b> is the handle of the user file to close, as returned by <b>UserFileOpen</b> .  |
| <b>Return Value</b> | If the user file was closed successfully, this function returns “1”. Otherwise, the return value is “0”.  |
| <b>Description</b>  | <p>This function must be called at the end of your REXX program to close the user file. Failure to call <b>UserFileClose</b> may cause data to be lost.</p> <p>After calling <b>UserFileClose</b>, the handle returned by <b>UserFileOpen</b> is no longer valid.</p> |

---

## UserFileCreateRecord

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | rc=UserFileCreateRecord( <b>huf</b> )   |
| <b>Arguments</b>    | <b>huf</b> must be a user file handle returned by <b>UserFileFindOpen</b> .   |
| <b>Return Value</b> | This function returns “1” if the user record was created successfully, or “0” otherwise.  |
| <b>Description</b>  | <p>The “usr.” stem variable must be filled out (as a minimum) with the new user's name and alias. The user record in this structure will be added to the specified user file as a new user record.</p> <p>Ideally, your program should first call the <b>UserFileInitRecord</b> function to initialize the “usr.” stem variable. Next, your program should call <b>UserFileGetNewLastread</b> to obtain a new lastread pointer for the user. Finally, your program should fill out the “usr.name” and “usr.alias” variables, along with any other desired settings.</p> |

---

## UserFileFind

---

|                  |  |
|------------------|--|
| <b>Prototype</b> | rc=UserFileFind( <b>huf</b> , <b>name</b> , <b>alias</b> )   |
| <b>Arguments</b> | <p><b>huf</b> specifies a user file handle, as returned by <b>UserFileOpen</b>.</p> <p><b>name</b> specifies the name of the user to find. If this field is blank, the user's name is ignored when trying to find a match.</p> |

**alias** specifies the alias of the user to find. If this field is blank, the user's alias is ignored when trying to find a match.

**Return Value** This function returns “1” if the specified user was found; otherwise, it returns “0”.

**Description** When **UserFileFind** successfully finds a user, it places all of the information related to that user in the “usr.” stem variable. For information on accessing these fields, please see section 9.3.

This function tries to find a single user in the user file. If both **name** and **alias** are non-blank, **UserFileFind** only tries to find a user record that matches both the name and alias fields.

If **name** is specified but **alias** is an empty string, **UserFileFind** ignores the alias field and only tries to find a user whose name matches **name**.

If **alias** is specified but **name** is an empty string, **UserFileFind** ignores the name field and only tries to find a user whose alias matches **alias**.

If both **name** and **alias** are empty strings, **UserFileFind** returns the first user in the user file.

---

---

## UserFileFindClose

---

---

**Prototype** rc=UserFileFindClose(huff)

**Prototype** **huff** is a user file find handle returned from **UserFileFindOpen**.

**Return Value** This function returns “1” if the user find handle was closed successfully, or “0” otherwise.

**Description** This function must be called to deallocate the memory and files associated with a user file finding operation.

---

---

## UserFileFindNext

---

---

**Prototype** rc=UserFileFindNext(**huff**, **name**, **alias**)

**Arguments** **huff** is the user file find handle, as returned by UserFileFindOpen.

**name** specifies the next name to look for in a sequential search of the user file. A blank string matches the next available user name.



**alias** specifies the next alias to look for in a sequential search of the user file. A blank string matches the next available user alias.

|                     |  |
|---------------------|--|
| <b>Return Value</b> | If a user matching the specified criteria was found, this function returns “1” and stores the user's information in the “usr.” stem variable. If no user was found, this function returns “0”.             |
| <b>Description</b>  | This function returns the next user in a <b>UserFileFindOpen</b> search. The <b>name</b> and <b>alias</b> parameters do not need to be the same as specified in the original <b>UserFileFindOpen</b> call. |

---

### UserFileFindOpen

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>huff = UserFileFindOpen(huff, name, alias)</code>   |
| <b>Arguments</b>    | <b>huff</b> is the user file handle returned by <b>UserFileOpen</b> .   |
| <b>Arguments</b>    | <b>name</b> is the name of the user to find, or a blank string to match any user name.<br><br><b>alias</b> is the alias of the user to find, or a blank string to match any alias.  |
| <b>Return Value</b> | If a user matching the name/alias criteria is found, this function returns a positive handle for the search operation. If no matching users were found, this function returns “-1”.   |
| <b>Description</b>  | If a user is found, the “usr.” stem variable is filled out with the information in the user record. The <b>UserFileFindNext</b> function can be called to find subsequent users.<br><br>The <b>name</b> and <b>alias</b> parameters should be specified using the same rules as given for <b>UserFileFind</b> . |

---

### UserFileFindPrior

---

|                  |  |
|------------------|--|
| <b>Prototype</b> | <code>rc=UserFileFindPrior(huff, name, alias)</code>   |
| <b>Arguments</b> | <b>huff</b> is the user file find handle, as returned by <b>UserFileFindOpen</b> .<br><br><b>name</b> specifies the next name to look for in a reverse sequential search of the user file. A blank string matches the user prior to the current record.<br><br><b>alias</b> specifies the next alias to look for in a reverse sequential search of the user file. A blank string matches the user prior to the current record. |

|                     |  |
|---------------------|--|
| <b>Return Value</b> | If a user matching the specified criteria was found, this function returns “1” and stores the user's information in the “usr.” stem variable. If no user was found, this function returns “0”. |
| <b>Description</b>  | This function performs the same function as <b>UserFileFindNext</b> , except that the search starts at the current user record and searches backward to the beginning of the user file.        |

---

## UserFileGetNewLastread

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | usr.lastread_ptr = UserFileGetNewLastread( <b>huf</b> )                                    |
| <b>Arguments</b>    | <b>huf</b> must be a user file handle returned by <b>UserFileOpen</b> .                    |
| <b>Return Value</b> | This function returns the next available lastread pointer in the user file.                |
| <b>Description</b>  | This function should be used to obtain a lastread pointer when creating a new user record. |

---

## UserFileInitRecord

---

|                    |  |
|--------------------|--|
| <b>Prototype</b>   | call UserFileInitRecord( <b>huf</b> )  |
| <b>Arguments</b>   | <b>huf</b> must be a user file handle returned by a call to <b>UserFileOpen</b> .  |
| <b>Description</b> | This function initializes the REXX “usr.*” stem variable to a set of standard values. This function can be called before filling out a user record for a new user. |

---

## UserFileOpen

---

|                  |  |
|------------------|--|
| <b>Prototype</b> | huf = UserFileOpen( <b>userfile</b> , <b>mode</b> )  |
| <b>Arguments</b> | <p><b>userfile</b> is the path and root filename of the Maximus user file, without the extension. In a typical Maximus installation, this field is “\max\user”.</p> <p><b>mode</b> is the file opening mode to use when opening the user file. If <b>mode</b> is “create” the user file will be created only if it does not exist. Otherwise, if the mode is not “create” (or if the user file already exists), the existing user file is opened in read/write mode.</p> |

|              |   |
|--------------|---|
| Return Value | On success, <b>UserFileOpen</b> returns a handle for the user file. This handle must be passed to all future <b>UserFile*</b> functions. If the user file could not be opened, this function returns “-1”.  |
| Description  | <p>This function open the Maximus user file for interaction. <b>UserFileOpen</b> must be called before accessing any of the other REXX API functions.</p> <p>The user file may not necessarily be called <b>user.bbs</b>. It is best to allow the name of the user file to be passed as an argument to your REXX program.</p> <p>This function must be called before any of the other user file functions can be used. The value returned by this function should be stored in a variable so that it can be passed as the argument to the other <b>UserFile*</b> functions.</p> |

---

---

### UserFileSize

---

---

|              |  |
|--------------|--|
| Prototype    | size = UserFileSize( <b>huf</b> )  |
| Arguments    | <b>huf</b> is the user file handle, as returned by the <b>UserFileOpen</b> call                                |
| Return Value | This function returns the number of users in the user file, or -1 if an invalid user file handle was provided. |
| Description  | This function counts the number of users in the specified user file.   |

---

---

### UserFileUpdate

---

---

|              |   |
|--------------|---|
| Prototype    | rc=UserFileUpdate( <b>huf</b> , name, alias)  |
| Arguments    | <p><b>huf</b> is a user file handle returned from <b>UserFileOpen</b>.</p> <p><b>name</b> is the name of the user to modify. This parameter is not optional.</p> <p><b>alias</b> specifies the alias of the user to modify. This parameter is not optional.</p> |
| Return Value | This function returns “1” if the update was successful, or “0” otherwise.   |
| Description  | This function takes the user information stored in the “usr.” stem variable and writes it into the user record for the named user. If the specified user does not exist, this function will fail.   |

The **name** and **alias** parameters must specify the name and alias of the user, as it currently exists in the user file. In most cases, these will simply be “usr.name” and “usr.alias”. However, in cases where the REXX program modifies the user’s name or alias, these fields must point to the user record’s original name and alias.

### 9.3. Accessing “usr.” Variables

All of the fields in the user record are exported to the REXX namespace using the “usr.” stem variable. Most of these variables are exported as strings, but some are stored as binary variables.

Table 9.2 lists the types that are used to store various parts of the user record.

**Table 9.2 REXX Types**

| Type   | Description   |
|--------|---|
| string | This variable represents a variable-length string. If a maximum length is imposed on the string, this is indicated in square brackets beside the type name. |
| char   | This variable represents an 8-bit integer. Any number in the range -128 to 127 is valid.  |
| int    | This variable represents a 16-bit integer. Any number in the range -32768 to 32767 is valid.  |
| long   | This variable represents a 32-bit integer. Any number in the range -2147483648 to 2147483647 is valid.  |
| bool   | This variable represents a boolean value. The character “Y” represents YES; the character “N” represents NO. No other values are valid.                     |
| priv   | This variable represents a privilege level. Values in the range 0 through 65535 are valid.  |
| help   | This variable represents a help level. This field can contain any of the following values: “Novice”, “Regular”, or “Expert”.                                |
| video  | This variable represents a video mode. This field can contain any of the following values: “TTY”, “ANSI”, or “AVATAR”.                                      |
| date   | This variable represents a date. If the date is modified, it must be stored in exactly this format:   |

```
dd mmm yy hh:mm:ss
```

**dd** is the two-digit, zero-padded day of month (01-31).

**mmm** is the three-character abbreviation for the month name. This field must be one of “Jan”, “Feb”, “Mar”, ..., “Dec”.

**yy** is the last two digits of the current year (“95”).

**hh**, **mm** and **ss** represent the hours, minutes and seconds in 24-hour time.

All of these two-digit numbers should be zero padded such that “1” through “9” become “01” to “09”.

For example: “09 Jul 95 06:15:02”

**keys** This variable contains a list of keys held by the user. All of the standard Maximus key values can be enabled by simply listing those values in the string. All keys from 1 to 8 and A to X are valid.

The keys need not be placed in the string in any particular order. For example, if a user had keys A, B, F, 3 and 6, the keys string could be set to “ABF36”. Any other permutation (such as “6FB3A”) is also treated identically.

---

Table 9.3 lists the fields made available to the REXX program after a call to **UserFileFind**, **UserFileFindNext**, or **UserFileFindPrior**. All of these variables are stored under the “usr.” stem variable. For example, to access the “city” field, your REXX program must reference “usr.city”.

**Table 9.3 REXX User Record Structure**

| Name         | Type       | Description  |
|--------------|------------|--|
| num          | int        | The user number of the current user. This variable must not be modified.                               |
| name         | string[35] | The primary log-on name of the given user.   |
| city         | string[35] | The user’s “city” or location.   |
| alias        | string[20] | The user’s alias or secondary log-on name.   |
| phone        | string[15] | The user’s phone number.   |
| lastread_ptr | int        | The user’s lastread pointer offset. For new users, this field can be filled in by calling <b>User-</b> |

**FileGetNewLastread.**

|               |            |  |
|---------------|------------|--|
| timeremaining | int        | The number of minutes that the user had remaining when he/she last logged off. |
| pwd           | string[15] | This is the user's password.   |

**WARNING!** If the “encrypt” field (below) is set to “Y”, the password field is stored as a 16-byte encrypted string which cannot be displayed. Otherwise, if the “encrypt” field is set to “N”, the password is stored in plaintext and can be displayed as a normal string.

There are no facilities available in REXX for manipulating encrypted passwords, so care should be taken when updating this field.

To assign a new password to a user, set the “pwd” field to the plaintext password, and ensure that the “encrypt” field is set to “N”.

Note that Maximus will automatically convert the password to an encrypted format the next time it accesses the user record.

|          |       |  |
|----------|-------|--|
| times    | int   | The number of times that the user has called the system.             |
| help     | help  | The user's current help setting.                                     |
| video    | video | The user's current video setting.                                    |
| nulls    | char  | The number of NULs sent after each end-of-line character.            |
| hotkeys  | bool  | The user's hotkeys setting.  |
| notavail | bool  | If this is set to “Y”, the user cannot be paged for multinode chats. |
| fsr      | bool  | If this is set to “Y”, the full-screen reader is enabled.            |
| nerd     | bool  | If this is set to “Y”, the user's yells are silenced.                |

|            |      |  |
|------------|------|--|
| noulist    | bool | If this is set to “Y”, the user will not be displayed in the standard user list.   |
| tabs       | bool | If this is set to “Y”, groups of 8 spaces will be sent to the user as tabs.  |
| encrypt    | bool | If this is set to “Y”, the user's “pwd” field is currently stored in an encrypted format. Otherwise, the password is stored as plaintext.  |
| rip        | bool | If this is set to “Y”, <i>RIPscrip</i> graphics are enabled.   |
| badlogon   | bool | If this is set to “Y”, Max will behave as if the user entered an incorrect password on the last log-on attempt.  |
| ibmchars   | bool | If this is set to “Y”, Max will send high-bit IBM extended ASCII characters directly to the user. Otherwise, it will attempt to translate those characters to 7-bit equivalents. |
| bored      | bool | If this is set to “Y”, the BORED line editor is enabled. Otherwise, the MaxEd full-screen editor is enabled.   |
| more       | bool | If this is set to “Y”, the user will receive “More [Y,n,=]?” prompts after every page of output.   |
| configured | bool | If this is set to “Y”, Maximus has already asked the user the introductory log-on questions, such as “Use ANSI?”, “Use MaxEd?”, and so on.                                       |
| cls        | bool | If this is set to “Y”, the user wants screen-clearing codes to be sent.  |
| priv       | priv | The user's current privilege level.  |
| time       | int  | The number of minutes for which the user was on-line, between 00:00 and 23:59, on the date specified by <code>usr.ludate</code> .  |
| deleted    | bool | If this flag is set to “Y”, the user is marked as deleted.   |
| permanent  | bool | If this flag is set to “Y”, the user is marked as permanent. (This means that the user cannot be   |

|           |      |   |
|-----------|------|---|
|           |      | deleted by automated purging programs.)   |
| width     | char | The width of the caller's screen.   |
| len       | char | The height of the caller's screen.  |
| credit    | int  | The user's credit value for NetMail messages.   |
| debit     | int  | The user's debit value for NetMail messages. The user's NetMail balance can be calculated using this formula:<br><br>balance = credit - debit                           |
| xp_priv   | priv | The privilege level to which the user will be demoted when the user's subscription expires. The user will only be demoted to this value if usr.expdemote is set to "Y". |
| xp_date   | date | The date by which the user's account will expire. This value will only be checked if usr.expdate is set to "Y".   |
| xp_mins   | long | The number of on-line minutes remaining in the user's subscription. This field is only debited if usr.expmins is set to "Y".  |
| expdate   | bool | If this is set to "Y", the expire-by-date feature is activated.   |
| expmins   | bool | If this is set to "Y", the expire-by-time feature is activated.   |
| expdemote | bool | If this is set to "Y", the user is demoted to the privilege level specified by the xp_priv field when the subscription expires.   |
| expaxe    | bool | If this is set to "Y", Maximus will hang up and purge the user's user record when the subscription expires.   |
| ludate    | date | The beginning time of the user's last call.   |
| xkeys     | keys | The keys held by the user.  |
| lang      | char | The user's current language number, with 0 being the first language declared in the language con-   |



trol file, 1 being the second language, and so on.

|              |            |   |
|--------------|------------|---|
| def_proto    | char       | This represents the user's default protocol. A value of -1 indicates no default protocol; -2 is Xmodem; -3 is Ymodem; -4 is Xmodem-1K; -5 is SEAlink; -6 is Zmodem; and -7 is Ymodem-G.<br><br>A value of 0 or greater represents an external protocol. |
| up           | long       | The total number of kilobytes uploaded by the user.   |
| down         | long       | The total number of kilobytes downloaded by the user.   |
| downtoday    | long       | The number of kilobytes downloaded by the user today.   |
| msg          | string     | The user's current message area.  |
| files        | string     | The user's current file area.   |
| compress     | char       | The user's default compression format. A value of -1 indicates no default compressor.   |
| dataphone    | string[18] | The user's business/data phone number.  |
| dob_year     | int        | The year in which the user was born.  |
| dob_month    | int        | The month in which the user was born.   |
| dob_day      | int        | The day of month in which the user was born.  |
| msgs_posted  | long       | The number of messages that the user has posted.  |
| msgs_read    | long       | The number of messages that the user has read.  |
| sex          | int        | The user's gender. (0 = unknown; 1 = male; 2 = female.)   |
| date_1stcall | date       | The date of the user's first call.  |
| date_pwd_chg | date       | The date of the user's last password change.  |
| nup          | long       | The number of files that the user has uploaded.   |

## 154 9. REXX User File Interface

|              |      |  |
|--------------|------|--|
| ndown        | long | The number of files that the user has downloaded.          |
| ndowntoday   | long | The number of files that the user has downloaded today.    |
| time_added   | int  | The time credits given to the user for the current day.    |
| point_credit | int  | The number of points that have been credited to the user.  |
| point_debit  | int  | The number of points that have been debited from the user. |

---

---

# 10. Introduction to MEX Programming

## 10.1. About MEX

The Maximus Extension Language (MEX) is an advanced programming language designed to interface with the Maximus bulletin board system. MEX is a true programming language that provides support for many advanced language features, including functions, dynamic strings and arrays.

MEX can be used to add user-defined features and extend the functionality of a standard Maximus system. For example, MEX can be used to implement a full-screen chat program, a call-back verifier, or to implement a user-specific database using standard file I/O function calls.

MEX was designed to overcome limitations in MECCA, which was the original Maximus extension language. MECCA was primarily intended to handle simplistic screen formatting and graphics, but it could also handle very simple forms of flow control and menus.

Although MECCA is still supported, MEX is the extension language of choice for many advanced tasks. As an example of MEX's power, parts of the standard Maximus distribution have been rewritten in MEX, including the file and message area headers, the Change Menu, and more.

MEX can interface to internal Maximus routines to obtain information about the current user, to display screen output, or to perform repetition and flow control. In fact, MEX programs can accomplish almost anything that can be done in a general-purpose programming language such as C or Pascal.

However, all of this power does not mean that MEX is hard to use. The MEX language incorporates the best features from the C, BASIC and Pascal languages, including time-saving features such as dynamic strings, pass-by-reference parameters, and structures.

## 10.2. MEX Road Map

The rest of this section introduces a simple MEX program, describes how to compile MEX programs, and describes how to run MEX programs from Maximus.

To find more information about MEX:

- For general information on MEX programming and language features, please see section 11.
- For information on the MEX run-time library, please see section 15.
- For reference information on the MEX language itself, including the language grammar, please see section 16.

## 10.3. Creating a Sample MEX Program

Like MECCA, MEX is a *compiled* language. This means that the description of what the program does, otherwise known as the *source file*, must be processed by the MEX compiler before the program can be run. Source files always have an extension of *.mex*.

The source for a MEX program is simply an ASCII text file. An external editor, such as the DOS *edit.com* or the OS/2 *e.exe*, is used to create and edit a MEX source file.

The best way to start programming in MEX is to write a sample program. To do this, start up a text editor (such as one of the two programs mentioned above). Then enter the following lines, using punctuation and spaces exactly as shown below:

```
#include <max.mh>

// This is a hello world program.

int main()
{
    print("Hello, world\n");
    return 0;
}
```

After the text has been entered, double-check each line to ensure that it appears just like given above.

When run, this program will simply print “Hello, world” and return to Maximus. However, before trying to compile and run this program, it will help to become familiar with the basic elements of the MEX language:

```
#include <max.mh>
```

This line, “`#include <max.mh>`”, must be present in all MEX programs. This line instructs the MEX compiler to read in a `.mh` file (a **MEX header**) that tells it how to interface with Maximus.

Without the **max.mh** file, MEX would not know how to display output to the user, how to access information in the user record, or how to perform many other operations.

Unless otherwise noted, **max.mh** is the only header file that needs to be included in most MEX programs.

```
// This is a hello world program.
```

This line is a comment. A comment begins with two forward slashes, and everything on the same line, following the slashes, is ignored by the compiler.

The MEX compiler completely ignores comments, so it is quite possible to write a “commentless” program. However, comments serve to remind the reader about how a program works, and they can also act as an aid to other people who try to modify an existing MEX program.

In general, it is a good idea to place comments throughout a MEX program, especially in areas where the program logic is not self-evident.

```
int main()
{
    (inner portion of source omitted)
}
```

The structure shown above is known as a *function*. Functions are the building blocks which are used to create MEX programs. All of the actions that a MEX program performs are contained within functions. Only a brief description of functions will be given here, but a detailed discussion of functions can be found in section 11.

The first line of the function, “`int main()`”, tells the MEX compiler to create a function called “main”. All MEX programs must have a **main** function. When a MEX program receives control from Maximus, the program will always start running at the **main** function.

The braces, “`{`” and “`}`”, serve as bounds for the **main** function. Everything that is part of the **main** function must be inside the braces.

```
print("Hello, world\n");
```

This statement is the “meat” of our simple program, since it instructs Maximus to perform a specific action. This line is a *function call*, which consists of a *function name*, followed by a pair of parentheses which contain the *function arguments*.

In this case, the function name is **print**. The **print** function is used to send output to the screen.

The function arguments are “Hello, world\n”. The **print** function will take the provided arguments and display them on the screen. (The “\n” is a newline escape sequence, as will be explained later in more detail.)

In short, while the function name tells Maximus *what* to do, the function arguments tell Maximus *how* to do it.

Several other things are noteworthy about this function call:

- The line ends with a semicolon. In MEX, all statements **must** end with a semicolon. In most places, the MEX compiler ignores whitespace (such as spaces, tabs, and ends of lines), so it uses the semicolon to mark the end of a statement.
- Notice that the function argument, “Hello, world\n”, is enclosed in double quotes. In MEX, this is known as a *string*. Strings consist of zero or more letters, numbers, or punctuation marks that are surrounded by a pair of double quotes. Strings are used for storing, manipulating, and displaying words and other types of textual information.
- In the example above, the **print** function was only called with one argument. Some functions only accept a fixed number of arguments, but in the case of *print*, it can accept as many arguments as desired.

To pass more than one argument to a function, simply add a comma-delimited list of arguments within the parentheses. For example, to print the words “Alex”, “Betty” and “Chuck”, the following line could be used:

```
print("Alex ", "Betty ", "Chuck ");
```

In addition, some functions do not accept any arguments. Even in these cases, the parentheses must still be present, since they denote that a function call should be made. For example, given a function called **foo** that accepted no arguments, it could be called like this:

```
foo();
```

- In the function argument, the last two characters, “\n,” are known as an *escape sequence*. When the MEX compiler sees a backslash followed by the letter *n*, it recognizes this as the *newline escape sequence*. This escape sequence then gets

translated to a linefeed, which causes the cursor to skip down to the next line after the string is displayed.

In short, every time MEX sees a “\n” enclosed in double quotes, the “\n” causes the cursor to be sent to the beginning of the next line.

For example, if our sample program contained the following line:

```
print("This is a\n test of the\n\n output text.\n");
```

then the output would look like this:

```
This is a
test of the

output text.
```

While other escape sequences are also supported, only the newline escape sequence is important for now.

We finish our analysis of the sample program by examining the last line in the *main* function:

```
return 0;
```

This line is a *return statement*. This statement indicates to Maximus that the MEX program completed successfully. This normally appears as the last line in the **main** function. (The return statement has other uses which will be discussed later; but for simple, single-function programs, only one return statement is necessary.)

This completes our analysis of the sample MEX program. See the following sections for information on compiling and executing MEX programs under Maximus.

## 10.4. Compiling MEX Programs

The main purpose of the MEX compiler, **mex.exe**, is to read in a source **.mex** file and compile it into a **.vm** file. This **.vm** file can be read and executed by Maximus. The compilation converts the MEX source file into a form that can be executed while a user is on-line.

The secondary purpose of the MEX compiler is to check the program source for errors. If the compiler detects an error while compiling your source, it will print an error message and abort the compilation. A text editor must then be used to edit the source file and correct the error.

Assuming that you called your file **test.mex**, simply type the following to compile the program:

```
mex test.mex
```

The compiler will then process the source file. If the source file is incorrect, the compiler will display error messages and abort the compilation. If this happens, ensure that the source file matches the example given in the section above. (The line numbers given in the warning messages can be used to determine the approximate location of the error.)

## 10.5. Running MEX Programs

Once a MEX program has been compiled, it can be executed by Maximus.

There are three primary ways to invoke a MEX program from Maximus: from a menu option, from a **.bbs** or **.mec** file, or from a **File** or **Uses** statement in **max.ctl**:

### 10.5.1. Running MEX Programs from Menu Options

The easiest way to add a new MEX program to the system is to give it its own menu option. To add a menu option, a line similar to the following can be added to one of the menu definitions in **menus.ctl**.

For example, the following line can be added to the definition for “Menu Main” to run the MEX program that was created in the previous section:

```
MEX      m\test      Normal      "Test "
```

This line instructs Maximus to execute a program called **m\test**. Maximus will assume that the file is relative to the main Maximus directory, so if you installed Maximus in **c:\max**, it will expect to find the file in the **c:\max\m** directory.

After making changes to **menus.ctl**, the Maximus control files should also be re-compiled with SILENT.

Maximus reads in the compiled source file (as generated by the MEX compiler), so it will actually be looking for a file called **c:\max\m\test.vm**. If the file is not found, an appropriate error message will be displayed (and also placed in the system log).

If everything went well, upon pressing the “T” key at the main menu, the output of the “Hello world” program should appear on the screen.

For more information on the format of menu options, please see section 18.8.



**10.5.2. Running MEX Programs from .MEC Files**

If desired, MEX programs can also be run directly from **.mec** or **.bbs** files. The *[mex]* MECCA token tells Maximus to run a MEX program:

For example, these lines could be added to **welcome.mec**:

```
[mex]m\test
[pause]
```

After recompiling the **welcome.mec** file with MECCA, the **c:\max\m\test.vm** program will be run every time a user logs on, in addition to displaying the rest of the normal welcome screen.

**10.5.3. Running MEX Programs from MAX.CTL**

In many of the Maximus control files, numerous keywords allow **.bbs** filenames to be specified. Maximus allows a MEX program to be substituted for a **.bbs** file in any of these places.

To indicate to Maximus that it is to run a MEX program instead of a **.bbs** file, simply add a colon (":") to the beginning of the filename. For example, to completely replace the **\max\misc\logo.bbs** file with a MEX program, the **Uses Logo** statement in **max.ctl** can be modified to read as follows:

```
Uses Logo      :M\Test
```

In this line were used, instead of displaying the standard **\max\misc\logo.bbs** file, Maximus would instead run the **\max\m\test.vm** program.

This technique can also be applied to any other part of the system that requests a **.bbs** filename. For example, even though the **MEX** menu option can be used to run MEX programs directly, we can also use the **Display\_File** menu option to display a MEX program, even though it is normally used to display **.bbs** files:

```
Display_File    :M\Test           Normal    "Test "
```



---

# 11. MEX Language Tutorial

This section is an introduction to programming in MEX. No prior programming experience is assumed, although familiarity with other programming languages will help in some areas. While this section tries to teach some basic programming principles, it is not a substitute for a general text on programming methodology.

Before reading this section, new programmers should first read and try out the “Hello world” program from the introduction..

Those who are already familiar with C or Pascal should read section 12. That section provides a brief overview of the differences between MEX and each of the other two languages. However, material in this section will also be helpful in demonstrating basic language principles.

## 11.1. Program Development Cycle

Like many things, designing a MEX program is an iterative task. Seldom do developers write a program correctly on the first attempt; this section is an introduction to the development cycle of a typical computer program.

The first step in designing a computer program is to determine the end goal of the program. What must the program do? What are its inputs? What are its outputs?

These three questions must be answered before any further steps can be taken in program design. Without a clear idea of what a program is supposed to accomplish, proceeding to the design and implementation stages all but guarantees a poorly-designed program in the long run.

After the program’s inputs and outputs have been determined, the next step is to decompose the program’s operations into functional units. This process is described in more detail in section 11.2.1 (below), but in essence, it involves breaking a large problem into many smaller subproblems.

The next step is the implementation stage. Here, you must enter the program source code (or a portion thereof) into the computer. The program source code now takes the form of an ASCII source file with a **.mex** extension.

Now, the MEX compiler is invoked to compile the program source. If any compilation errors occur, you must go back to the previous step and re-edit the program.

Additionally, a compile-time error might alert you to a problem with the program design. In this case, you must return to the design stage and modify the program specification.

Next, the compiled MEX program is run from within Maximus. Even if the program compiles correctly, there still may be numerous functional errors in the program. If problems are observed, you must return to a previous stage, either to re-edit the program source, or to re-design the portion of the program that is causing the problem.

In theory, if a program is designed correctly the first time, very few design changes will be required in the later development stages. However, as a program grows in size, it becomes much more difficult to design the program in sufficient detail to ensure that all of the program components will interoperate correctly without any design revision.

In addition, other factors may necessitate a design change, such as users requesting additional features to be added to your program.

This entire development cycle is often referred to as the “waterfall effect.” Problems noted in the implementation stage may require a return to the design stage; problems in the compilation stage may require a return to the implementation *or* design stage; problems in the execution stage may require a return to any of the preceding stages.

In addition, the impact of “feature requests” should not be underestimated. A common sentiment among programmers is that no one else except themselves will use a program, so why bother to design it or comment it properly? Unfortunately, these types of programs all too often end up being distributed and used by others, so it usually pays to design and implement a program correctly the first time.

For programs with a large base of installed users, the program development cycle never stops. Small changes are always being made to the program design; small tweaks are always being made to the program source code; and minor problems are always being found during program execution.

Familiarizing yourself with the program development cycle is the best way to ensure success when developing a program, no matter how large or how small.

## 11.2. Elements of a MEX Program

A standard MEX program consists of two main components: a set of instructions that tell the computer what to do, and the information on which those instructions operate.

The set of instructions is often referred to as *code*, while the information on which it operates is often referred to as *data*.

### 11.2.1. About Code

In MEX, the program code is implemented using *functions*. Although the sample program in the introduction had only one function, **main**, most MEX programs have a number of interconnected functions. Functions are building blocks that are used to design programs in a modular manner.

For example, given a MEX program that retrieves a user's date of birth, the program could be broken up into a number of functions:

- One function could display a prompt and ask the user to enter a line of text.
- Another function could validate the text entered to ensure that it is a valid date.
- A third function could store the date information in the user record.
- A fourth and final function, the **main** function, could tie all of the above functions together by invoking them in the right order.

Within each function are a number of *statements*. These statements are the step-by-step instructions that tell the computer what to do. For example, the statements in the "date validation function" could do something similar to the following:

- Separate the date into year, month and day components
- Check if the year is greater than 1890. If not, reject the date.
- Check if the month number is greater than 0 and less than 13. If not, reject the date
- Check if the day number is within the bounds for the indicated month. (1 to 31, 1 to 30, 1 to 28, and so on)

By breaking down the major components of a program into functions, and then by breaking down each function into a sequence of statements, programs can be written to solve complex everyday problems.

**11.2.2. About Data**

Program data consists of *constants* and *variables*. Like the name implies, constants never change. Given an expression such as “ $2 + 2$ ”, the number “2” is a constant.

Variables, on the other hand, can change while your program is being run. Variables must be given names, as in conventional mathematics. For example, given an expression such as “ $i + 2$ ,” the symbol  $i$  is a variable. Depending on the value of  $i$ , the expression itself can have different values. For example, if  $i$  has a value of 6, the value of the expression “ $i + 2$ ” is 8.

In MEX, a variable also has a *type*. The type of a variable indicates the kind of information that the variable is allowed to hold. Table 11.1 lists some of the basic data types supported by MEX:

**Table 11.1 MEX Data Types**

| Type   | Description  |
|--------|--|
| char   | Variables of this type can hold a single character, including letters, numbers, digits, punctuation, and control characters. Variables of this type can also be used to store very small numbers in the range of 0 to 255. Examples of characters are ‘a,’ ‘6,’ and ‘:’. Characters are described in more detail in a later section. |
| int    | An int can hold numbers in the range -32768 to 32767. This is an “integer” data type, meaning that it can only hold whole numbers, both positive and negative. This means that numbers that contain a fractional component, such as 8.6 and -4.3, are not allowed.   |
| long   | A long can hold numbers in the range -2147483648 to 2147483647. This is an “integer” data type, meaning that it can only hold whole numbers, both positive and negative. This means that numbers that contain a fractional component, such as 8.6 and -4.3, are not allowed.   |
| string | This is a “dynamic string” data type. Strings are commonly used for many word or text-oriented applications, such as printing output to the screen, reading a sequence of keystrokes from the keyboard, or any other task which requires characters to be “grouped.”   |

Strings are dynamic in nature, meaning that they grow and shrink along with their contents. Consequently, no “maximum length” need be set when declaring a string.

Examples of dynamic strings are “Maximus” and “Alexander the Great.”

Strings are described in more detail in a later section.

|        |  |
|--------|--|
| struct | A structure is an aggregate, user-defined data type that contains multiple objects of other types. For example, a structure could hold an integer, a character and a string. Structures will be discussed in more detail in a later section. |
| array  | An array is an aggregate, user-defined data type that contains multiple objects of the <i>same</i> data type. For example, an array could contain 500 distinct integers.   |
| void   | This is a special data type that is normally only used for functions that do not return values. The void data type cannot hold a value and cannot be used in calculations. This type is discussed in more detail in a later section.         |

---

The three integral types described above, **char**, **int** and **long**, come in both *signed* and *unsigned* versions. A signed type can represent a negative number, whereas an unsigned type cannot. (However, an unsigned type can store numbers twice as large as a signed type.)

The modifiers **signed** and **unsigned** can be inserted in front of one of the three integral type names to specify a signed or an unsigned type.

The **char** type is unsigned by default, whereas **int** and **long** are both signed by default.

For example, MEX supports the **signed char** type (with a range of -128 to 127), the **unsigned int** type (with a range of 0 to 65536) and the **unsigned long** type (with a range of 0 to 4294967296).

### 11.3. Variable Declarations

All MEX programs must *declare* variables before they can be used. The variable declaration tells the compiler two things: the name of the variable, and the type of information that the variable will be used to store.

In a MEX program, variables can be declared in two different places:

- In a *block* at the top of the **.mex** source file. Variables declared here are known as *global variables*. Global variables are accessible to all of the functions in a program, and these variables maintain their value throughout the execution of the entire program.

- In a *block* at the top of each function. Variables declared here are known as *local variables*. Local variables are only accessible within the function in which they are declared. These variables lose their values as soon as the containing function finishes executing.

A variable *block* simply denotes an area of the source code that contains one or more variable declarations. There is no explicit keyword to tell the MEX compiler that it is processing a variable block or that variables are being declared; this is detected by context alone. Variables can be declared at the top of the source file, and just after the opening brace (“{”) of functions.

Note to advanced programmers: variables can also be placed at the beginning of any *basic block* in a function, and at other top-level points in the source file (outside of functions). However, this capability should only be used when absolutely necessary.

A variable declaration block looks like this:

```
int: i, j, k;
```

The word **int** is the variable type. (In declarations, everything to the left of the colon (“:”) is treated as the variable type.)

The comma-delimited list that follows the type, “i, j, k,” is the list of variables to declare.

The variable declaration ends with a semicolon (“;”). This semicolon tells the compiler where the declaration ends.

For the most part, variable names can be assigned arbitrarily. However, a few restrictions are imposed:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that “delta,” “Delta,” “DELTA,” and “DeLtA” refer to four distinct objects.
- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that “top10” is a valid name, whereas “7up” is not.)

In the example above, three integers were created, with names *i*, *j* and *k*. Of course, variables of different types can also be declared within one block. The example below shows how to declare variables of different types within a declaration block:

```
char: c;
string: str;
int: i;
```



```
int: j;
```

In the declaration above, a character *c* is created, in addition to a string *str* and integers *i* and *j*. Notice that a semicolon follows each declaration.

A comma-delimited list could have also been used to write an equivalent declaration block:

```
char: c;
string: str;
int: i, j;
```

As can be seen above, variables of the same type can be declared in a comma-delimited list, as well as in separate declarations, with no change in functionality.

All integral variables are automatically initialized to 0 when they are declared. All string variables are initialized to the null string ("").

### 11.3.1. Character Variables

Both the **char** and **string** data types are used for storing displayable information. However, characters and strings are used for quite distinct purposes.

A **char** is normally used when only one letter, number, punctuation symbol or control character needs to be stored. In addition, a **char** can store any integer number that is in the range 0 to 255. Hence, **chars** are also useful for storing very small numbers.

To assign a value to a variable of type **char**, simply enclose the letter, number or punctuation mark in single quotes. For example:

```
char: c;

c := 'q';           // c holds the letter q
      or
c := ':';           // c holds a colon
```

This approach is adequate for assigning most types of character constants. However, some character values cannot be conveniently entered in this approach. For example, how does one enter a newline character, or for that matter, how does one enter a character that contains a single quote?

The answer lies in *escape characters*. Within character constants (and also within strings), the *backslash* is used as an escape character. Whenever the compiler recognizes an escape character in a character or string constant, it also reads the immediately-following character and treats the two as a pair.

By using two-character escape sequences, you can easily place non-printable and special characters in the source file. Table 11.2 lists the escape characters supported by MEX:

**Table 11.2 MEX Escape Characters**

| Escape character | Name            | ASCII code | Description  |
|------------------|-----------------|------------|--|
| <code>\n</code>  | Newline         | 10         | As seen in the sample MEX program in the introduction, this character sends the cursor down one line and moves it to the left-hand side of the screen. |
| <code>\r</code>  | Carriage return | 13         | A carriage return moves the cursor to the beginning of the current line.   |
| <code>\a</code>  | BEL             | 7          | The BEL character sends an audible beep to the user. (When displayed using <b>print</b> , this is normally not heard on the local console.)            |
| <code>\b</code>  | Backspace       | 8          | The backspace character moves the cursor back by one column.   |
| <code>\f</code>  | Formfeed        | 12         | The formfeed character clears the screen.  |
| <code>\t</code>  | Tab             | 9          | The tab character sends the cursor to the next horizontal tab stop.  |
| <code>\'</code>  | Single Quote    | 39         | A single quote character (').  |
| <code>\"</code>  | Double Quote    | 34         | A double quote character (").  |
| <code>\\</code>  | Backslash       | 92         | A backslash character (\).   |

### 11.3.2. String Variables

Strings are normally used when a large number of characters need to be manipulated as a single block, or when parts of a block of characters need to be manipulated independently.

Strings can be assigned just like any other variables:

```
string: s;

s := "Julius Caesar";
```

After this code is run, the variable *s* will contain “Julius Caesar.” To modify the string at a later point in the program, only reassigning the string is necessary, as shown below:

```
#include <max.mh>
```

```

int main()
{
    string: s;

    s := "Napoleon";

    // ... perform conquests here ...

    s := "King Henry IV";
    return 0;
}

```

In the above program segment, the assignment of strings of varying length to *s* is automatically managed by the MEX run-time environment. No explicit “maximum length” need be specified when declaring the variables, regardless of the size of the string.

Strings can include any of the escape characters described in the previous section.

In addition, MEX allows you to manipulate parts of strings as characters. The “[ ]” operator can be used to extract a character from a specific point in a string, or to place a new character into a string. The expression:

```
str [ idx ]
```

instructs MEX to extract character number *idx* from the string and to convert it to a character, with an *idx* of 1 representing the first character in the string.

This notation can be used for both examining characters in strings and modifying the characters in an existing string. MEX will automatically expand the string and pad it with spaces if an expression attempts to add a character beyond the end of the string.

For example:

```

#include <max.mh>

int main()
{
    string: s;
    char: c;

    s := "Philistine";
    c := s[1];           // c is now 'P'
    s[3] := c;           // s is now "PhPlistine"

    // Now expand string by writing beyond end.

    s[13] := 'o';        // s is now "PhPlistine o"
    print(s);
}

```

```
        return 0;  
    }
```

## 11.4. Variable Scope

A variable's *scope* is defined as the region of the source code which is capable of accessing or modifying that variable. The MEX rules for scoping are derived from the C language (which was in turn derived from Algol 68).

In MEX, two simple scoping rules are applied:

- The scope of global variables begins at the point in the source file where they are declared. The scope of the variable extends to the end of the file. A variable which is declared as global can be used by any function in the entire program, and the variable's contents are preserved across function calls. Variables can be declared globally by simply placing the declaration at the top of the file, outside of any function body.
- The scope of a local variable begins at the opening brace (“{”) of the block in which it is declared. The scope extends over all nested blocks, and it terminates at the closing brace (“}”) of the block in which it was declared. Variables declared inside a function are said to have a local scope, since they can only be accessed from within that function. Variables with a local scope are automatically allocated and deallocated when a function is called; hence, local variables do not have their values preserved across different calls of the same function.

## 11.5. Preprocessor

The MEX compiler includes a rudimentary preprocessor which is controlled by *preprocessor directives*. These directives instruct it to perform compile-time substitutions and comparisons of keywords in the source code.

All preprocessor directives begin with a hash (“#”) character at the beginning of the line.

MEX supports the following preprocessor directives:

```
#define name value
```

The **#define** directive instructs the compiler to examine the source code and replace all instances of the word *name* with the value of *value*. For all intents and purposes, the compiler behaves as if *value* had been entered in the source code rather than *name*. All such substitutions are performed before a source line is processed by the rest of the compiler.

For example, the following piece of code:

```
#define ASSIGN_VARIABLE myint

int: myint;

ASSIGN_VARIABLE := 3;
```

is equivalent to the following code in all respects:

```
int: myint;

myint := 3;
```

```
#include <filename>
```

The **#include** directive instructs the compiler to read in the header file *filename* and process it as if its contents were placed directly within the source file being compiled. This directive is useful for splitting up a large file into multiple parts.

In the introduction, the **#include** directive was also used to include a file called **max.mh**, the standard system include file. The **max.mh** file contains all of the definitions required to interface your program with Maximus. While these definitions could be copied into each and every MEX program, it is much more convenient to put these definitions all in one place and then **#include** the file from all MEX programs.

The filename to be included must be placed within angle brackets, as shown in the example above. MEX will first look for the file in the current directory; if it cannot be found there, MEX will try to find the file in the same directory as the **.mex** file that is currently being compiled. Finally, MEX will attempt to find the file in the directory specified by the **MEX\_INCLUDE** environment variable.

## 11.6. Calculations and Arithmetic

MEX supports a wide variety of arithmetic and logical operators which permit many different operations to be performed on the variables and data used by a program.

After declaring a set of variables that can hold arbitrary values, the next thing a programmer may want to do is assign values to those variables.

The simplest way to assign a value to a variable is to use the assignment operator. The assignment operator is entered as two separate characters: a colon followed immediately by an equals sign (“:=”).

The assignment operator is similar to the “equals sign” in conventional mathematics; it assigns the *value* on the right-hand side of the assignment operator to the *variable* on the left-hand side.

For example, after running the following program:

```
#include <max.mh>

int main()
{
    int: i;
    char: c;
    long: l;

    i := 4;
    c := 'q';
    l := 12345678;

    return 0;
}
```

the variable *i* will contain the number 4, *c* will contain the character 'q', and *l* will contain the number 12345678.

One variable can also be assigned to another, as shown in the following program:

```
#include <max.mh>

int main()
{
    int: i, j;

    i := 3;        // i now contains 3
    j := i;        // j now contains 3 too

    return 0;
}
```

In addition, many other operations can be performed on variables. Many of the standard arithmetic operations, including addition, subtraction, multiplication and division, can also be performed on variables of types **char**, **int** and **long**. (Note that arithmetic operations cannot be performed on strings. However, the “+” symbol functions as a *catenation operator* to combine the contents of two strings.)

MEX also supports a set of boolean logic operators, including “and” and “or.” Lastly, MEX also supports bitwise logic operators, such as *bitwise and* and *bitwise or*.

In general, an operator can be combined and used with other variables as follows:

*<expression-left> <operator> <expression-right>*

*<operator>* is one of the arithmetic operators from the table below. *<expression-left>* and *<expression-right>* can be arbitrary expressions made up of other operators. For an example, given “3 + 5,” the *<expression-left>* would be “3,” the *<operator>* would be “+,” and the *<expression-right>* would be “5.”

Table 11.3 lists the operators supported by MEX:

Table 11.3 MEX Operators

| Operator     | Description   |
|--------------|---|
| +            | <i>Addition and catenation.</i> This operator yields the arithmetic addition of two integer operands, such as an <i>int</i> and an <i>int</i> , or a <i>long</i> and a <i>long</i> .<br><br>Also, when both <i>&lt;expression-left&gt;</i> and <i>&lt;expression-right&gt;</i> are strings, the + operator functions as a catenation operator to combine the two strings. |
| -            | <i>Subtraction.</i> This operator yields the arithmetic difference of the right-hand operand and the left-hand operand.   |
| *            | <i>Multiplication.</i> This operator yields the arithmetic multiplication of the left-hand operand with the right-hand operand.   |
| /            | <i>Division.</i> This operator yields the arithmetic division of the left-hand operand by the right-hand operand.   |
| %            | <i>Modulo-division.</i> This operator yields the remainder of the left-hand operand divided by the right-hand operand.  |
| =            | <i>Equality.</i> The result of an equality comparison is 1 if the two compared objects contain the same value, or 0 if they do not.   |
| <>           | <i>Inequality.</i> The result of an inequality comparison is 1 if the two compared objects do not contain the same value, and 0 if they do.   |
| <=, <, >=, > | <i>Logical comparators.</i> These are the <i>less-than-or-equal</i> , <i>less-than</i> , <i>greater-than-or-equal</i> and <i>greater-than</i> operators. All of these operators yield a result of 1 if the comparison between the left-hand operand and the right-hand operand is true; otherwise, they yield 0.  |
| and          | <i>Logical and.</i> The <i>logical and</i> operator yields a result of 1 if both of its operands are non-zero; otherwise, it yields 0.  |
| or           | <i>Logical or.</i> The <i>logical or</i> operator yields a result of 1 if either (or both) of its operands are non-zero; otherwise, it yields 0.  |
| &            | <i>Bitwise and.</i> This operator yields the <i>bitwise and</i> of two integral operands. Each bit in the result is a 1 only if the bits in the equivalent positions in both operands are <i>both</i> equal to 1.   |



|     |  |
|-----|--|
|     | <i>Bitwise or.</i> This operator yields the <i>bitwise or</i> of two integral operands. Each bit in the result is a 1 if <i>either</i> of the bits in the equivalent positions of the operands are equal to 1.   |
| shl | <i>Shift-left.</i> This operator shifts the left-hand operand to the left by the number of bits specified by the right-hand operand. This operator implements a logical shift and does not necessarily preserve the sign of the operand, regardless of whether it is signed or unsigned. |
| shr | <i>Shift-right.</i> This operator shifts the left-hand operand to the right by the number of bits specified by the right-hand operand. This operator implements a logical shift; bits shifted in on the left-hand side of the word are always 0.   |

---

The normal mathematical order of evaluation applies to all calculations. (A table listing the exact operator precedence can be found in section 16.)

When expressions involve operands which have different types, the operand with the smallest range is always promoted to the type of the other operand. If two operands are the same type but one is **unsigned** and the other is **signed**, the **unsigned** operand is always converted to a **signed** operand.

This code demonstrates the simple arithmetic operators:

```
#include <max.mh>

int main()
{
    int: i, j, k;

    i := 3;
    j := 4 * i;      // j = 12
    k := j / 3 + 1;  // k = 5

    return 0;
}
```

In addition, parentheses can be used in any expression to explicitly specify the order of evaluation:

```
#include <max.mh>

int main()
{
    int: i, j, k;

    i := 3;
    j := 4 * i;      // j = 12
    k := j / (3 + 1); // k = 3
}
```

```

    return 0;
}

```

The logical operators are also quite useful for evaluating true/false expressions. Unlike other languages, MEX has no explicit “boolean” data type. Instead, any integral expression can be evaluated as a boolean expression. Expressions which evaluate to zero are considered to be *false*, while all non-zero expressions are considered to be *true*. Boolean-like macro definitions for TRUE and FALSE can be found in **max.mh**.

For example:

```

#include <max.mh>

int main()
{
    int: i, j, k, l, m;

    i := 0;
    j := 4;

    k := i and k;           // k = 0
    l := i or j;            // l = 1
    m := l and k;           // m = 0

    return 0;
}

```

Although MEX does not have an explicit *not* operator, it is easy to test if a boolean condition is false by simply comparing it with 0:

```

#include <max.mh>

int main()
{
    int: i, j, k, l;

    i := 2;
    j := 3;
    k := (i = 3);           // k = 0 since i <> 3
    l := (k = 0)            // l = not k = 1

    return 0;
}

```

Last but not least, the + operator can be used to catenate (combine) strings. For example:

```

#include <max.mh>

```

```

int main()
{
    string: s1, s2, s3;

    s1 := "Foo";
    s2 := "Bar";
    s3 := s1 + s2;           // s3 = "FooBar"
    print(s3);
    return 0;
}

```

The string catenation operator can also be applied to multiple strings at a time:

```

#include <max.mh>

int main()
{
    string: s1, s2, s3, s4;

    s1 := "string1";
    s2 := "string2";
    s3 := "string3";
    s4 := s1 + s2 + s3;
    print(s4);
    return 0;
}

```

## 11.7. Displaying Output

As seen in the sample program in the introduction, the **print** function is used to display output to the user. However, **print** is not limited to displaying just predefined text. The **print** function can also display the contents of variables and combine multiple data types into one statement.

In general, multiple arguments can be used within a **print** statement by providing a comma-delimited list of variables or constants to be displayed. Providing such a list is equivalent to listing each variable in a sequence of separate **print** statements.

In other words, given a declaration such as this:

```

string: a, b, c;

a := "Alpha ";
b := "Bravo ";
c := "Charlie ";

```

the following code segment:

```

print(a, b, c);

```

will display “Alpha Bravo Charlie”. However, the following code segment will also produce the same output:

```
print(a);
print(b);
print(c);
```

However, **print** is not limited to displaying only strings. Both characters and integers can also be displayed:

```
#include <max.mh>

int main()
{
    char: c;
    int: i;
    string: s;

    s := "The best solution to the problem is ";
    c := '#';
    i := 1;

    print(s, c, i);
    return 0;
}
```

This code will display:

```
The best solution to the problem is #1
```

One further point is that the **print** function does not add any extra spacing when displaying variables. Had an extra space not been included at the end of the string *s*, the output would have appeared as follows:

```
The best solution to the problem is#1
```

Similarly, the **print** function does not automatically move the cursor to the next line after displaying the specified variables. If this cursor movement is desired, the newline escape sequence ('\n') should be inserted at the end of the print statement, as shown below:

```
string: s;
char: c;

s := "Text #";
c := '2';

print(s, c, '\n');
```

This code will display “Text #2” and move the cursor to the beginning of the next line.

## 11.8. Flow Control

In the preceding MEX examples, the programs have all executed the statements in the **main** function from top to bottom. However, we may want to execute only some of the statements enclosed in a function, or in some cases, we may want to repeat some of its statements more than once.

The *flow* of a program is the order in which the computer executes statements within a function. Adding *flow control* statements to a MEX program allows the designer to modify the flow and specify the conditions necessary for groups of statements to be executed or iterated (repeated).

There are two types of flow control statements: conditional statements and iterative statements. Conditional statements allow a set of statements to be executed or skipped based on a specified condition, while iterative statements allow a group of statements to be executed multiple times.

### 11.8.1. Conditional Execution

The most basic form of conditional flow control is the **if** statement. The **if** statement allows the conditional selection of a group of statements.

The **if** statement has two separate forms. The standard form looks like this:

```
if (expr)
    statement1
```

If the expression inside the parentheses evaluates to non-zero, *statement1* is executed. Otherwise, *statement1* is skipped.

The second form of the **if** statement looks like this:

```
if (expr)
    statement1
else
    statement2
```

This is also referred to as the **if / else** statement. If *expr* evaluates to non-zero, only *statement1* is executed. Otherwise, only *statement2* is executed.

Note that **if** itself is a statement, so multiple **if / else** statements can be chained together as shown below:

```

if (a+b > c)
    print("a+b > c");
else if (b+c < a)
    print("b+c < a");
else if (d*e = 0)
    print("d*e = 0");
else
    print("none of the above");

```

In addition, the **if** statement can be used to conditionally execute multiple statements by using a *compound statement*. A compound statement is a pair of braces, between which can be any number of other statements. This means that constructs such as this can be used:

```

if (a > c)
{
    print("a > c\n");
    d := 0;
}
else
{
    print("a <= c\n");
    d := 1;
}

```

In this example, if *a* is greater than *c*, MEX will display “a > c” and set *d* to 0. Otherwise, it will display “a <= 3” and set *d* to 1.

The **if** statement can also be nested, as shown below:

```

if (a > c)
{
    if (a > d)
        print("a is bigger than c and d\n");
    else
        print("a is bigger than c only\n");
}
else
    print("a is smaller than c\n");

```

### 11.8.2. Iterative Flow Control

Iteration is used when one or more statements need to be executed multiple times. Iteration is also known as looping or repetition.

MEX supports three different types of iterative flow control: the **while** statement, the **do .. while** statement, and the **for** statement:

**11.8.2.1. while**

The simplest form of iteration is the **while** statement (or **while** loop). This statement instructs MEX to repeat the following statement as long as a given expression is non-zero.

A standard **while** statement looks like this:

```
while (expr)
    statement1;
```

As with the **if** statement, *statement1* can be either a single statement or a compound statement.

Before every iteration of the loop, including before the first iteration, *expr* is evaluated. If **expr** evaluates to zero, the loop is skipped. Otherwise, *statement1* is executed once, and then the whole process repeats. (This implies that if *expr* is initially zero, the loop will never be executed.)

For example, the following code fragment causes the body of the **while** loop, the “*i* := *i* + 1” statement, to execute ten times:

```
#include <max.mh>

int main()
{
    int: i;

    i := 1;

    while (i <= 10)
        i := i + 1;

    return 0;
}
```

The “*i* := *i* + 1” could have easily been replaced with a compound statement, such as one which increments the counter and then calls **print** to display the value of the counter:

```
#include <max.mh>

int main()
{
    int: i;

    i := 1;

    while (i <= 10)
```

```

    {
        print("i = ", i, '\n');
        i := i + 1;
    }

    return 0;
}

```

#### 11.8.2.2. do .. while

Conceptually, the **do .. while** statement (or **do .. while** loop) is very similar to the **while** statement. A **do .. while** statement looks like this:

```

do
    statement1
while (expr);

```

The only difference between a **do .. while** statement and a standard **while** statement is the point in time at which *expr* is evaluated. With a **do .. while** statement, the statement *statement1* is executed first, and then *expr* is tested. This means that the loop will always be iterated at least once, regardless of the value of *expr*.

```

#include <max.mh>

int main()
{
    int: i;

    i := 0;

    do
    {
        print("i = ", i, '\n');
        i := i + 1;
    }
    while (i > 0 and i <= 9);

    return 0;
}

```

This code will increment *i* from 1 to 10, as in the previous example. However, the loop condition has been modified so that the loop will exit if “*i* > 0 and *i* <= 10” is false.

On the first iteration, *i* is set to zero. However, the loop condition is not tested until the end of the loop, after *i* has been incremented to 1, so the loop condition is true.



**11.8.2.3. for**

The **for** statement (or the **for** loop) is also quite similar to the **while** statement, but the **for** statement allows the loop initialization and termination conditions to be written in a much more concise manner.

A **for** statement looks like this:

```
for (initexpr; testexpr; postexpr)
    statement1;
```

When processing a **for** loop, the compiler will proceed as follows:

1. Before anything else is done, *initexpr* is evaluated. Since this expression is evaluated only once, it is commonly used to initialize a variable used in the loop test. (Note that “expressions” such as this can include assignments and function calls.)
2. Next, *testexpr* is evaluated. If this expression evaluates to zero, the loop is skipped.
3. *statement1* is executed. This statement is also known the *body* of the loop. Overall, *statement1* is continually executed as long as *testexpr* evaluates to non-zero.
4. *postexpr* is evaluated. This expression is normally used to increment some sort of loop counter.
5. Branch back to step 2.

A simple **for** statement looks like this:

```
for (i := 1; i <= 10; i := i + 1)
    print("i = ", i, '\n');
```

This loop simply counts from one to ten, displaying the value of *i* while doing so. To break down the loop operation in detail:

1. Before the loop begins, *i* is initialized to 1.
2. Next, *i* is compared with ten. This test succeeds, so the print statement in the body of the loop is executed.
3. *i* is then incremented by one.
4. The whole process repeats by comparing *i* with ten, and then performing the actions from 2 through 4 again.

From this, one can see that the **for** statement is just a shorthand notation for the **while** statement. An equivalent representation for the **for** statement is shown below:

```
initexpr;

while (testexpr)
{
    statement1;
    postexpr;
}
```

### 11.8.3. Goto and Labels

There are few cases when the preceding flow-control statements cannot be used to implement an arbitrary program structure. However, for those rare cases when the preceding statements are not sufficient, MEX provides a **goto** statement.

The **goto** statement unconditionally transfers program control to a specific point in the program. The **goto** statement can be used to jump to another statement that either precedes or follows the **goto** statement itself, as long as the destination of the jump is within the same function as the **goto**.

The **goto** statement is often said to be the antithesis of structured programming, but when used very sparingly, it can sometimes simplify error-handling or allow the programmer to exit a deeply-nested loop.

A **goto** statement looks like this:

```
goto label;
```

where *label* is the user-defined destination for the jump.

A label declaration looks like this:

```
labelname:
```

where *labelname* is an arbitrary name that follows these conventions:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that “delta,” “Delta,” “DELTA,” and “DeLtA” refer to four distinct objects.
- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that “top10” is a valid name, whereas “7up” is not.)

The *labelname* is used to specify the destination in the *goto* statement. When the program encounters a “goto *labelname*” statement, it will immediately jump to the statement following *labelname*.

The following code fragment demonstrates how **goto** is used to break out of a **for** loop:

```
#include <max.mh>

int main()
{
    int: i, sum;

    // Start sum off at zero.

    sum := 0;

    // Count from 1 to 10

    for (i := 0; i <= 10; i := i+1)
    {
        // Increment sum by the value of the counter.

        sum := sum+i;

        // If the count is at least 14, exit
        // the loop.

        if (sum >= 14)
            goto out;
    }

    out:
    print(sum);
    return 0;
}
```

There are much more elegant ways to implement this function, such as by moving the *sum* check into the *testexpr* of the **for** loop. Regardless, this code fragment shows how to add up all of the numbers from one to ten, but exiting the loop as soon as *sum* is greater than or equal to 14. (In the case above, *sum* will equal 15 when the loop finally terminates.)

## 11.9. Function Calls

As previously described in section 11.2.1, functions are the main building blocks of MEX programs. All MEX programs must contain a **main** function, but most pro-

grams will also have many other related functions. These functions are used to break a large problem down into smaller, more-manageable parts.

In addition to the functions defined in the **.mex** source file, a large number of external functions can also be called by a MEX program. These functions perform various actions such as: displaying output to the user, prompting the user for input, accessing the Maximus user file, manipulating the file tag queue, and more. The code for these functions is contained within Maximus itself, but the functions can be called just like ordinary MEX functions declared in the source file. (A complete list of external functions is contained in section 15.)

In this section, the *calling function* refers to the point in the code from which a function call is made. The *called function* refers to the destination of a function call, or the code that gets executed when the function call is performed.

This section describes how to call functions, how to write your own functions, and how to pass data from one function to another.

### 11.9.1. Calling Functions

A function call looks like this:

```
funcname(arglist);
```

*funcname* is the name of the function to call. This name simply identifies a function that has already been declared or defined in either the **.mex** source file or in the global **max.mh** header file.

*arglist* is an optional, comma-delimited list of *arguments* to be used in the function call. Arguments are data objects that are shared between functions. Arguments allow the calling function to communicate with the called function, and vice versa.

Although the comma-delimited list of arguments is optional, the pair of parentheses is not. Even if a function accepts no arguments, an empty pair of parentheses must follow the function name.

In the calling function, a function call can be written either as a separate statement (as shown above), or for functions that return values, the call can also be included in expressions. This is discussed in more detail in section 11.9.4.

Normally, arguments are used for one-way information transfer (to send information from the calling code to the function that is called), but if desired, the function arguments can also be used to transfer information in the other direction.

The function call syntax should already be familiar, since the **print** function (as was used in many of the code segments given in prior sections) is just one example of a function call.

When a MEX program encounters a function call, the following steps are taken:

1. The current program location is saved.
2. The arguments specified in *arglist* are copied so that the called function can access the information therein.
3. Control is transferred to the called function. The statements in the called function are executed until it issues a **return** statement or executes the very last statement in the function.
4. The program location is restored to the position saved in step 1.

For example, the following example demonstrates how to sequentially call a number of functions. (The declarations for these functions will be discussed in the following section.)

```
#include <max.mh>

// insert function definitions here

int main()
{
    initialize_data();
    open_file();
    write_file();
    close_file();
    deinitialize_data();
    return 0;
}
```

In this case, the **main** function is nothing but an empty shell that calls other functions. The above code instructs **main** to execute the statements in the **initialize\_data**, **open\_file**, **write\_file**, **close\_file** and **deinitialize\_data** functions, in the order specified above.

The called functions can also include function calls in their own code, and the functions called from there can also call other functions. In theory, an infinite number of function calls can be nested in this manner, but in practice, only a few dozen nested calls can be made at a time, depending on the size of the function arguments and the stack size specified on the MEX command line.

The **print** function is just one example of a function that requires arguments. Some functions can also accept multiple arguments, as shown below:

```

#include <max.mh>

// insert function definitions here

int main()
{
    int: sailors;
    int: yachts;

    sailors := 8;
    yachts := 2;

    initialize_data(sailors, yachts);

    sail_yacht_1(sailors / yachts);
    sail_yacht_2(sailors - (sailors / yachts));

    return 0;
}

```

The first two statements initialize the variables *sailors* and *yachts* to 8 and 2, respectively.

The next statement is a function call to **initialize\_data**. The two variables *sailors* and *yachts* are passed as arguments. In this hypothetical example, the **initialize\_data** function presumably uses these values to set up data structures describing the number of sailors and the number of yachts that are in use.

Next, the **sail\_yacht\_1** function is called, with a number representing about half of the sailors. The function could use that information to determine how many sailors were in the boat's crew.

Finally, the **sail\_yacht\_2** function is called, using the number of remaining sailors as an argument. The function would also presumably use the number of sailors to determine how it operates.

As can be seen from the example above, although functions are used to break up the code of a program into smaller parts, the function argument list is how information is passed from one function to another.

### 11.9.2. Defining Functions

Previous sections have focused on calling external functions, or as-of-yet undeclared and undefined functions in the user code. This section describes how to define functions of one's own that can be called from other points in the program.

A function definition has this form:

```

return_type funcname(paramlist)
{
    statements
}

```

This definition can be broken down as follows:

*return\_type* is the type used for the function's *return value*. The return value is an optional piece of information that is passed back to the calling function. (If desired, the function arguments can also be used to pass information back to the caller.) Not all functions have return values; when that is the case, the *return\_type* is "void."

*funcname* is the name of the function. This name is used when calling this function via a function call. Function names must abide by the following conventions:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that "delta," "Delta," "DELTA," and "DeLtA" refer to four distinct objects.
- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that "top10" is a valid name, whereas "7up" is not.)

*paramlist* is the optional, comma-delimited parameter list for the function. The parameter list consists of *pairs* of parameter types and parameter names. For functions that accept no parameters, the space within the parentheses must be left blank. The topic of function parameters is discussed in the following section in greater detail.

The pair of braces delimit the body of the function. The statements contained therein can be any of the statement types described in previous sections.

However, function definitions may not be nested. This means that function definitions must be placed at the "top level" of the source file, outside of any other function definitions.

A simple function definition looks something like this:

```

void hokey_pokey()
{
    print("bar\n");
}

```

This code defines a function called **hokey\_pokey**. This function can be called from the **main** function as shown below:

```

int main()
{

```

```
    print("foo\n");  
    hokey_pokey();  
    print("boz\n");  
  
    return 0;  
}
```

When run, this program will print:

```
foo  
bar  
boz
```

### 11.9.3. Function Prototypes

If the called function is not defined in the same program source file as the calling function (such as with external functions), or if the function is defined later in the same file, the compiler must be told some extra information about the called function, including its argument types and return value.

Regardless of whether a function is internal or external, the compiler needs some sort of information about the function — either a *function definition* or a *function prototype* — before that function can be called.

If an external function is called, a function prototype is required. If an internal function is called, and the *definition* of that function occurs at a later point in the source file than the function call itself, a function prototype is also required. (If the called function is defined in the same file, and the definition occurs above the point in the file where it is called, a prototype is not required. In this case, the function definition provides sufficient information to the compiler.)

A function prototype looks just like a function definition, except that the prototype has no function body. Instead, the statements and braces are replaced with a single semicolon.

A function prototype looks like this:

```
type funcname(paramlist);
```

*type* is the return type of the function. This type must match the return type given in the function definition.

*funcname* is the name of the function. This too should match the name of the function given in the function definition.



*paramlist* is the parameter list for the function. Except in special cases with variable-parameter functions, this list should also match the parameter list given in the function definition.

Function prototypes must be placed at the “top level” of the source file, outside of any other function definition, and before any attempt is made to call the function being prototyped.

Since a prototype tells the compiler about a function, before the definition for that function is presented, using prototypes allows two functions to call each other recursively:

```
#include <max.mh>

void func2();           // Prototype for func2

void func1()
{
    func2();            // Call func2, even though it has
                        // been declared. (The prototype
                        // above tells the compiler about
                        // func2 and the arguments/return
                        // value that it accepts.
}

void func2()
{
    func1();            // Call func1. A prototype is not
                        // required because the function
                        // was defined at an earlier
                        // point in the same file.
}
```

Note that for all of the external functions described in section 15, the appropriate function prototypes are already included in the **max.mh** include file. Hence, as long as the program contains the crucial “#include <max.mh>” line at the top of the file, all of the external functions will be automatically prototyped.

#### 11.9.4. Function Arguments

##### 11.9.4.1. Pass-By-Value Arguments

Up until now, functions have been treated as if they were simply logical blocks of a program, performing the same purpose each time they were called. Function arguments allow functions to perform different actions depending on the values of their arguments.

An important distinction must be made between function *arguments* and function *parameters*. From the *calling function*, the variables specified when writing the function call are known as *arguments*. Within the *called function*, these arguments are referred to as *parameters*. This implies that the *arguments* that are “sent” by the calling function are equivalent to the *parameters* that are “received” by the called function.

A function can accept as many parameters as desired; however, each parameter (and its associated type) must be specified when the function is defined and prototyped.

A definition for a function that has parameters looks like this:

```
type funcname(type1: var1, type2: var2)
{
    statements;
}
```

In this case, *type1* is the type of the first parameter, and *var1* is the name used to reference the parameter.

Similarly, *type2* is the type of the second parameter, and *var2* is the name used to reference the second parameter.

Naturally, more than two parameters can be used by simply expanding the comma-delimited list to provide more than two parameter type/name pairs. The comma-delimited list is also known as the function’s parameter list.

A special word of caution is required for function parameters:

All function arguments and parameters are *positional*. This means that the first parameter in a function definition must correspond to the first argument passed in a call to that function. Similar logic applies to the second and subsequent arguments and parameters.

Also, the parameter names specified in the called function are simply “dummy names.” The value to use for each parameter is provided by the argument list from the calling function. The *calling* function can pass any variable or expression for a given argument, but that value will become known to the *called* function by the parameter name specified in the function definition.

For example, given the following code:

```
#include <max.mh>

void myfunc(int: arg1, int: arg2)
{
    print("arg1 = ", arg1, " and arg2 = ", arg2, '\n');
}
```

```

int main()
{
    int: i, j;

    i := 4;
    j := 7;

    myfunc(i, j);
    return 0;
}

```

The program output will be:

```
arg1 = 4 and arg2 = 7
```

In the **main** function, the variable *i* is passed for *arg1*, and the variable *j* is passed for *arg2*. Within **myfunc**, the *arg1* and *arg2* names refer to the parameters specified in the call to that function, regardless of what those arguments were called in the calling function.

In **myfunc**, the value of *arg1* will be equal to the value of *i* (from the **main** function), and the value of *arg2* will be equal to the value of *j* (also from the **main** function).

However, within the called function, changes made to the function parameters do *not* automatically cause a change in the arguments specified in the calling function.

For example, given the following program:

```

#include <max.mh>

void change_it(int: a)
{
    a := 3;
}

int main()
{
    int: i;

    i := 5;
    change_it(i);
    print("i is ", i, '\n');
    return 0;
}

```

This program prints:

```
i is 5
```

As can be seen from the output, the assignment to *a* in **change\_it** has no effect on the value of *i* in **main**.

#### 11.9.4.2. Pass-By-Reference Arguments

MEX also supports a type of argument passing called *pass-by-reference*. Pass-by-reference arguments explicitly cause the argument passed into the function call to be modified when the parameter is updated from within the called function.

To declare a pass-by-reference argument in the function definition and prototype, simply include the **ref** keyword before the parameter type. For example, if the definition for **change\_it** were modified to look like this:

```
void change_it(ref int: a)
```

then the program would display:

```
i is 3
```

as could be expected. The only caveat to using pass-by-reference arguments is that a *variable* must always be specified for that argument when writing the function call, rather than using a constant or some other expression.

This means that, given the pass-by-reference definition of **change\_it**, the following function calls are illegal:

```
change_it(3);           // 3 is a constant
change_it(i * 2 + 2);   // i * 2 + 1 is an expression
```

A function with pass-by-reference parameters will update the function argument if the parameter is changed from within the called function. Obviously, the function cannot change the “value” of the number 3. Likewise, it also cannot change the “value” of the expression “*i* \* 2 + 1.”

Hence, arguments passed for pass-by-reference arguments must be simple variable names and not expressions or constants.

#### 11.9.5. Function Return Values

In all of the examples presented in previous sections, the functions had no *return value*. A return value is another method that can be used to pass information from the called function back to the calling function.

Function return values can be used in a manner similar to conventional mathematical functions. For example, if we had a function that calculated the square root of a number, one would like to be able to write:

```
i := sqrt(25);           // i = 5
```

To declare a function that returns a value:

1. In the function prototype and definition, select an appropriate return type for the function. This type must immediately precede the *funcname* part of the definition or prototype, as discussed in the preceding section.
2. Inside the function, the **return** statement must be used to identify the value being returned.

A return statement looks like this:

```
return opt-expr;
```

where *opt-expr* is an optional expression which has a type equivalent to the function return type.

In those special cases where the function is declared as returning **void**, meaning that the function does not return a value, the *opt-expr* must not be present. If that is the case, the return statement must look like this:

```
return;
```

For functions that are declared as returning **void**, the **return** statement simply instructs the function to return immediately to the calling function. No data is passed from the called function to the caller.

In cases where the function does return a value, upon encountering a **return** statement, the program will take the expression indicated in the **return** statement and store its value so that the calling function can access it. The program will then *immediately* return to the calling function.

For example, to implement the square root function from above:

```
int sqrt(int: value) // the "int" before the sqrt
{
    // identifies the type of
    // return value.
    int: result;

    // Code here that calculates the square root
    // of 'value' and places the result in 'result.'

    return result;
```

```
}
```

In this case, the function is said to **return** the value that is contained in the *result* variable. The naming of the variable used in the **return** statement is completely arbitrary; the variable can be named anything, just as long as the **return** statement indicates the value to be returned.

However, the **return** statement need not always be at the very end of the function. Since the **return** function causes the program to immediately end the function being called, the **return** statement is useful for error-checking and aborting a function before all statements have been executed.

For example, to add a degree of error-checking to our function:

```
#include <max.mh>

int sqrt(int: value)
{
    int: result;

    if (value < 0)
    {
        print("Invalid sqrt() argument!\n");

        return -1; // No real-valued number will
                  // have this root.
    }

    // Code to place the root of 'value' into 'result'.

    return result;
}
```

In the case shown above, if an invalid argument is passed to the **sqrt** function, it will print an error message and return a value of -1. Otherwise, it will proceed with the calculation and return the square root of the number.

From this, one can see that the **main** function from previous examples is an ordinary function that returns an integer value. The only difference between **main** and other functions is that Maximus will check the return value from **main**, and if it is non-zero, Maximus will place an entry in the system log. (**main** is also the function that Maximus first calls when a MEX program is started.)

However, functions are not constrained to returning only integers. Functions can also be used to return data of types **char**, **int**, **long** and **string**. For example, to define a function that returns a string describing the day of the week:

```
string weekday(int: daynum)
{
```

```

if (daynum = 1)
    return "Sunday";
else if (daynum = 2)
    return "Monday";
else if (daynum = 3)
    return "Tuesday";
else if (daynum = 4)
    return "Wednesday";
else if (daynum = 5)
    return "Thursday";
else if (daynum = 6)
    return "Friday";
else if (daynum = 7)
    return "Saturday";
else
    return "*InvalidDay*";
}

```

Compared to previous examples in this section, the only significant changes are that the function return value is declared as “string,” and the values following the **return** statements are all string constants.

The main restriction with return values is that only one data item can be returned to the caller. However, if multiple data items need to be passed back to the caller, pass-by-reference arguments are the obvious solution.

## 11.10. Arrays

Programs often need to manipulate large amounts of information in a similar manner. If a program needed to manipulate 500 numbers, the programmer could easily declare a variable to hold each individual number:

```

int: num1;
int: num2;
int: num3;
// and so on

```

However, if all of the numbers represented similar pieces of information (such as, say, the height of an office building), an *array* can greatly simplify the code used to manipulate this information.

An array is a data type that holds collections of other data objects. All of the objects contained within an array must be of the same data type.

In addition, the objects within the array can be *indexed*. For example, given an array of 500 numbers, the program can explicitly retrieve the 42nd number. Arrays can also be indexed by another variable. For example, the program can ask to re-

trieve the  $i$ th number from within the structure, and if the integer variable  $i$  happens to contain the value 3 at run-time, the third object in the array is retrieved.

Arrays can be considered as a group of data objects all lined up in a row. Each object in that row can be accessed individually by number, but the entire row can still be referred to as a single array of objects.

### 11.10.1. Declaring Arrays

An array declaration looks like this:

```
array [lower .. upper] of type: varname;
```

*lower* specifies the lower bound of the array. This is the lowest index value that can be used to store information in the array. Since most counting exercises start at the number one (as in *first*, *second*, *third*, and so on), this value is normally set to 1. (However, the code generated by the MEX compiler will run a little faster if the lower bound of the array is set to 0.)

*upper* specifies the upper bound of the array. This is the highest possible index value that can be used to store information in the array. The value used for *upper* depends on the number of data items that need to be stored in the array. *upper* must be greater than or equal to the value specified for *lower*.

All of the values between *lower* and *upper* (inclusive) can be used as indices for the array. In general, the total number of data objects stored in the array can be calculated as follows:

$$\text{Number of objects} = \textit{upper} - \textit{lower} + 1$$

A data object within an array is also sometimes referred to as an *element* in the array.

*type* is the data type of the objects contained in the array. As discussed earlier, all of the objects in the array will have the same type.

*varname* is the name of the array variable to be declared. *varname* must abide by the standard MEX variable-naming rules.

For example, this declaration creates an array of 100 integers:

```
array [1 .. 100] of int: my_ints;
```

In this case, the array variable is called *my\_ints*, and valid index values are 1 through 100.



Arrays of other data types can also be created; the indices for the array can also include negative values:

```
array [-50 .. 50] of string: my_strings;
```

Here, the array variable is called *my\_strings*, and valid index values are -50 through 50.

### 11.10.2. Accessing Arrays

Array variables themselves cannot be assigned or used in arithmetic expressions. However, the data objects within the array can be assigned to, manipulated, displayed, and treated like any other kind of variable.

After having declared an array, the *array operator* is used to access an individual data object within the array. Within an expression, the array operator is used like this:

```
arrayvar[idx]
```

*arrayvar* is the name of a previously-declared array variable, such as *my\_ints* or *my\_strings*.

*idx* is the index to be applied to the array, surrounded by square brackets. *idx* can be a constant (such as the number “4”), a variable (such as an integer), or the result of a more complex expression.

**This index value must always be within the bounds specified by *upper* and *lower* (from the array variable declaration). Using an out-of-bounds index value results in undefined behavior.**

When used with the array operator, the *arrayvar* and *idx* variables select a single data object from within the array. The array operator is always used as part of an expression. After applying the array operator, the resulting object can then be assigned to, manipulated as part of another expression, passed as a function argument, and treated like any other variable.

For example, given the *my\_ints* array declaration from above, the contents of the array can be set up as shown below:

```
my_ints[1] := 42;
my_ints[2] := 119;
my_ints[3] := 55;
// and so on
```

However, as mentioned previously, the array index need not be a simple constant, as in the “1, 2, 3” case above. To initialize all of the elements in the array to the value 42, a **for** loop can be used, along with an integer index:

```
#include <max.mh>

int main()
{
    array [1..100] of int: my_ints;
    int: i;

    // This 'for' loop increments 'i' from 1
    // to 100.

    for (i := 1; i <= 100; i := i + 1)
    {
        // Set each integer to 42.

        my_ints[i] := 42;
    }

    return 0;
}
```

A more complicated expression can also be used as an index. For example, to initialize only even-numbered data objects:

```
#include <max.mh>

int main()
{
    array [1..100] of int: my_ints;
    int: i;

    // Note that 'i' goes from 1 to 50, since we
    // are multiplying it by 2 below.

    for (i := 1; i <= 50; i := i + 1)
    {
        // Initialize data objects indexed by i*2,
        // which will result in indices of 2, 4,
        // 6, 8, and so on.

        my_ints[i*2] := 42;
    }

    return 0;
}
```

Similarly, the values in arrays can also be manipulated in other expressions. The following example creates an array containing the powers of 2:

```

#include <max.mh>

int main()
{
    array [1..8] of int: my_ints;
    int: i;

    for (i := 1; i <= 8; i := i + 1)
    {
        if (i = 1)
            my_ints[1] := 1;
        else
            my_ints[i] := my_ints[i-1] * 2;

        print("Array at ", i, " is ",
              my_ints[i], '\n');
    }

    return 0;
}

```

### 11.10.3. Arrays as Function Parameters

This section describes how to pass the information contained in arrays to other functions.

#### 11.10.3.1. Fixed-Length Arrays

To pass information in an array to a function, one possible option is to pass each element in the array as a separate argument, as shown below:

```

array [1..100] of int: my_ints;

// initialize array

process_an_int(my_ints[1]);

```

This example passes the first element of *my\_ints* to the **process\_an\_int** function.

However, the entire array can also be passed as a parameter in a function call. To do this:

1. Ensure that the array types match. In the function prototype and definition, the array parameter type must be identical to the argument type that the caller provides in the function call. For example, if passing an array called “my\_ints”, and if *my\_ints* has a type of “array [1..50] of string,” this exact type definition must also be included in the function prototype and definition.

2. Specify the name of the array variable when writing the function call. Just include the name of the array variable itself (without using the array operator).

For example:

```
#include <max.mh>

void handle_ints(array [1..50] of int: my_array);

int main()
{
    // Note that the type of the my_ints variable
    // matches the type in the function prototype
    // above.

    array [1..50] of int: my_ints;

    // initialize array here

    handle_ints(my_ints);
    return 0;
}

void handle_ints(array [1..50] of int: my_array)
{
    int: i;

    // Now loop through all of the integers and
    // display them.

    for (i := 1; i <= 50; i := i + 1)
        print("Int ", i, " is ", my_array[i], '\n');
}
```

The **handle\_ints** function takes the *my\_array* parameter and displays the value of every integer in the array.

Unlike other types, arrays are always passed by reference. Even if no **ref** qualifier is present in the parameter list, any changes that the *called* function makes to the array will also be reflected in the *caller's* copy of the array.

This allows a called function to adjust certain values in an array with little or no impact on program performance:

```
#include <max.mh>

void adjust_ints(array [1..50] of int: my_ints);

int main()
{
```

```

array [1..50] of int: foo;

// Initialize the array and display foo[2]

foo[1] := 3;
foo[2] := 8;
foo[3] := 9;

print("foo #2 is ", foo[2], '\n');

// Call adjust_ints and redisplay foo[2]

adjust_ints(foo);

print("foo #2 is ", foo[2], '\n');

return 0;
}

void adjust_ints(array [1..50] of int: my_ints)
{
    my_ints[2] := 4;
}

```

This program will display:

```

foo #2 is 8
foo #2 is 4

```

### 11.10.3.2. Variable-Length Arrays as Function Parameters

One common approach when using arrays is to write separate functions to manipulate the contents of an array of a specified type. Unfortunately, even though arrays containing the same type of object are used in many places, not all of these arrays are guaranteed to have the same bounds.

For example, suppose that we have two arrays containing the average daily temperature for each day in February and March (respectively). The arrays would be declared like this:

```

array [1..28] of int: feb_temp;
array [1..30] of int: mar_temp;

```

Now, suppose that we want to print out all of the temperatures for a given month. For our first attempt, we try creating a function with the following prototype:

```

int print_temps(array [1..30] of int: month_temp);

```

Unfortunately, even though we can call “print\_temps(mar\_temp)” using the above prototype, we get a MEX compile-time error if we try to call “print\_temps(feb\_temp).”

This error occurs because the bounds of the declared array do not match the bounds of the array specified in the function prototype. Separate **print\_temps** functions could be created with different array bounds, but this would lead to unnecessary duplication of program code.

To solve the problem in an elegant manner, MEX permits variable-length arrays in function prototypes and function definitions. The lower bound for the array must still be specified, and the lower bound must still match between the argument and the parameter. However, the upper bound can be omitted. This tells MEX that the called function can accept an array of any length.

A variable-length array can be declared as shown below. Note that this style of array declaration is only valid in function parameter lists and prototypes:

```
array [lower .. ] of type: varname
```

The only difference between this format and the standard array declaration format is the absence of *upper*. Aside from this omission, the components of the two declaration formats are identical.

Even though the *upper* bound can be omitted in the called function, it is still the called function’s responsibility to ensure that the array is only accessed within a valid set of bounds. The array referenced by the parameter is the same array as was declared in the calling function. Consequently, if the original array had bounds of 5 to 10, the called function must not use an index outside of the 5 through 10 range when accessing the parameter. The conventional way to deal with this problem is to have the calling function pass an extra argument that indicates the bounds of the array.

The following example shows how to use variable-length arrays to solve the “array of temperatures” problem discussed above:

```
#include <max.mh>

void print_temps(int: days,
                array [1..] of int: temps);

int main()
{
    array [1..28] of int: feb_temp;
    array [1..30] of int: mar_temp;

    // Initialize the February array
```

```

    feb_temp[1] := -15;
    feb_temp[2] := -13;
    // and so on

    // Initialize the March array

    mar_temp[1] := -3;
    mar_temp[2] := 2;
    // and so on

    // Now print them both out:

    print_temps(28, feb_temp);
    print_temps(30, mar_temp);
    return 0;
}

void print_temps(int: days, array [1..] of int: temps)
{
    int: i;

    // Loop through all of the days in the array,
    // as specified by the 'days' parameter, and
    // print out the temperature for that day.

    for (i := 1; i <= days; i := i + 1)
    {
        print("Temperature for day ", i, " is ",
              temps[i], '\n');
    }
}

```

### 11.11. Structures

When a function must manage a large amount of information, it is often helpful to group that information together in a logical manner. For example, to design an electronic address book, one possible (but clumsy) approach is to declare a separate variable for each item in the address book, like this:

```

string: name;
string: address;
string: city;
string: province;

```

Then, every time a data record needed to be read in or written out, each variable could be manipulated separately. Similarly, every time a function needed to access information in the address record, each variable would need to be passed as a separate parameter.

However, MEX provides a much easier way to group variables. The *structure* is an aggregate data type that acts as a “container” for other variables.

*Structure variables* (which are simply variables that are declared of the structure type) can then be manipulated, assigned and copied as a single data object. These actions act upon all of the fields contained within the structure. If desired, the fields within the structure can also be accessed individually.

Unlike arrays, structures can contain objects of different data types. Because of this, there is no way to specify an integer “index” to obtain a particular data object that is contained within the structure. (However, one can construct an array of structures; see section 11.11.4 below for more information.)

### 11.11.1. Defining Structure Types

Before a structure variable can be declared, the format of the structure itself must be defined. For example, in the address book example, we would need to tell the compiler that an “address book structure” contained the four string variables mentioned above.

The definition of a structure type looks like this:

```
struct tag
{
    field-decl-list
};
```

Structure definitions must be placed at the top of the source file, outside of any function definition.

*tag* is an arbitrary, user-defined “tag” for the structure. The *tag* identifies a particular structure type and its contents. A program can use many different structure types at the same time — such as both an address book structure and a phone book structure, as could be envisaged for our example above — so the tag is needed to differentiate between different structure types.

The tag can be set to any arbitrary name, as long as the name is unique among other functions, global variables and structures. The tag is used later when declaring variables of the specified structure type. The tag name must also abide by the usual variable-naming conventions:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that “delta,” “Delta,” “DELTA,” and “DeLtA” refer to four distinct objects.



- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that “top10” is a valid name, whereas “7up” is not.)

The *field-decl-list* contains a number of *field* definitions. A field is simply a data object contained within the structure. In our example at the beginning of the section, the *name* and *address* variables are fields in the address book structure.

A single semicolon must follow the closing brace of the structure.

The *field-decl-list* looks exactly like a variable declaration block at the beginning of a function. For example, if we were to give our address book structure a tag of “abook,” the definition would look like this:

```
struct abook
{
    string: name;
    string: address;
    string: city;
    string: province;
};
```

This structure definition has a tag of *abook*, and the structure contains the fields *name*, *address*, *city* and *province*.

However, as with variable declaration blocks, the fields in a structure need not all be of the same type. Our address book could be expanded to include more useful information, as shown below:

```
struct abook
{
    string: firstname;        // "John"
    char: initial;            // 'Q'
    string: lastname;         // "Public"

    int: house_number;        // 777
    string: street;           // Downing St.
    string: city;             // Kingston
    string: province;         // Ontario
};
```

This structure contains a number of *string* variables as before, but it also contains an *int* and a *char* to store individual pieces of the address information.

This structure type definition will be used by several examples in the following sections. This type definition should be included in any source file that uses the *abook* structure.

### 11.11.2. Declaring Structure Variables

Once a structure type has been defined, variables of that structure type can be created. Just like a normal `int` (or other variable type), structure variable declarations are placed in the variable declaration block at the beginning of a function.

A structure variable declaration looks like this:

```
struct tag varname;
```

The *tag* refers to a structure tag that was previously declared in a structure type definition.

*varname* is the name of the structure variable to be declared.

Given our address book example, suppose that we wanted to create two entries in the address book. We could then declare two structure variables as follows:

```
struct abook: john;  
struct abook: steve;
```

This would create variables called *john* and *steve*, each containing distinct copies of the fields in the address book structure.

### 11.11.3. Using Structure Variables

For the most part, structure variables can be manipulated like other data types. While it is obviously not possible to use arithmetic operators on entire structures (what would “steve + john” mean?), these structure variables can be assigned, passed as parameters in function calls, and so on.

In addition to the standard operators, the *field selector operator* (“.” — a period) is a special operator that can only be used with structures. This operator is used to directly access a single field contained within a structure.

Within an expression, the field selector operator is used like this:

```
structvar . field
```

*structvar* is the name of a previously-declared structure variable, such as “john” (from our previous example).

Following the period (“.”), the *field* is the name of one of the fields declared in the original structure definition, such as “firstname” or “street.”

By applying the field selector operator to a structure variable and a field name, single fields within the structure can be manipulated individually.

The field selector operator is always used as part of an expression. In other words, after applying the field selector operator to a structure variable and a field name, the result is always assigned to another variable, used as the target of an assignment, or included in some other expression.

For example, given an address book structure named *john*, we can assign a value to one of its fields like this:

```
john.lastname := "Public";
```

It may be useful to think of the structure operator as a way to select a certain field from a structure (which can then be used in the following operation).

After using the field selector, the result is an object which has a type equivalent to the type of the field. With our address book example, if we had declared a structure variable called *john*, the object “john.firstname” has a type of *string*. This means that *john.firstname* can be manipulated just like any other string.

In addition to assigning a string to *john.lastname*, as we did above, the following type of manipulation is also perfectly acceptable:

```
fullname := john.firstname + " " + john.lastname;
```

This expression takes the *firstname* field (which is a string) from within the *john* structure, and it then uses the catenation operator to add a space, followed by the contents of *john.lastname*.

The following sample program shows how values can be assigned to and retrieved from fields in a structure:

```
#include <max.mh>

struct abook
{
    string: firstname;
    char: initial;
    string: lastname;

    int: house_number;
    string: street;
    string: city;
    string: province;
};

int main()
```

```

{
    struct abook: john;
    struct abook: steve;

    john.firstname    := "John";
    john.initial      := 'Q';
    john.lastname     := "Public";
    john.house_number := 777;
    john.street       := "Downing St";
    john.city         := "Kingston";
    john.province     := "Ontario";

    steve.firstname   := "Steve";
    steve.initial     := 'S';
    steve.lastname    := "Smith";
    steve.house_number := 5;
    steve.street      := "Smith St";
    steve.city        := "Smiths Falls";
    steve.province    := "Ontario";

    print("John's last name is ",john.lastname,'\n');
    print("Steve's last name is ",steve.lastname,'\n');

    return 0;
}

```

This sample program should print:

```

John's last name is Public
Steve's last name is Smith

```

#### 11.11.4. Advanced Structure Definitions

Structures can be used to store simple data types, such as **int**, **char**, **string** and **long**, but structures can also be used to store arrays and even nested copies of other structures.

Structures containing other structures or arrays are declared in the same manner as less-complicated structures. In the case of structures containing structures, as long as the “child” structure (to be included within the “parent” structure) is defined at an earlier point in the source file (relative to the parent definition), the structure definition can be written as one might expect:

```

struct date
{
    int: year, month, day;
};

struct addressbook2

```

```

{
    string: name;
    struct date: date_entered;
    struct date: date_updated;
    array [1..2] of string: phone;
};

```

This example uses a child structure, *date*, contained within the parent structure, *addressbook2*. The parent structure contains two copies of the date structure, called *date\_entered* and *date\_updated*. It also includes an array of strings for storing data and FAX phone numbers.

The structure member operator (“.”) and array operators (“[” and “]”) can be combined in a logical manner to access the fields within these structures.

The address book structure from above can be declared and used like this:

```

struct addressbook2: john;

john.name := "John Q. Public";    // standard field

john.date_entered.year := 1995; // nested field
john.date_entered.month := 6;    // nested field
john.date_entered.day := 18;     // nested field

john.date_updated.year := 1995; // nested field
john.date_updated.year := 7;    // nested field
john.date_updated.day := 14;    // nested field

john.phone[1] := "+1-613-389-8315"; // array field
john.phone[2] := "+1-613-634-3058"; // array field

```

Other advanced constructs can also be used. For example, an array of structures can be declared to hold an entire address catalog. Fields within the structures can then be accessed like this, assuming the *addressbook2* definition from above:

```

array [1..10] of struct addressbook2: book;

book[1].name := "Steve S. Smith";
book[1].date_entered.year := 1995;
book[1].phone[1] := "+1-613-634-3058";
// etc.

book[2].name := "John Q. Public";
book[2].date_entered.year := 1995;
book[2].date_updated.month := 6;
// etc.

```

This makes it possible to perform the same operations on many structures, which has many applications in data processing programs. For example, given the address

book definition from above, the following program segment could be used to print out the names of everyone in the address book:

```
int: idx;

for (idx := 1; idx <= 10; idx := idx + 1)
{
    // If the name in this address book record is
    // not empty, print it out.

    if (book[idx].name <> "")
        print("Name is ", book[idx].name, '\n');
}
```

#### 11.11.5. Structures as Function Parameters

Structures can also be passed as parameters to functions. Just like other variable types, the contents of the structures can be accessed from within the called function using standard syntax.

In the function prototype and function definition, simply include the variable type and name in the parameter list. (For example, “struct addressbook: mybook” could be added to the parameter list to specify an address book structure.)

Next, when writing the function call, just include the name of the structure variable to be passed to the caller. For example:

```
#include <max.mh>

struct date
{
    int: year, month, day;
};

// Prototype for the function declared below.

void showdate(struct date: d);

int main()
{
    struct date: d1, d2;

    // Initialize dates

    d1.year := 1995;
    d1.month := 7;
    d1.day := 1;

    d2.year := 1994;
    d2.month := 3;
```

```

    d2.day := 24;

    // Display them to the user

    showdate(d1);
    showdate(d2);

    return 0;
}

void showdate(struct date: d)
{
    print(d.month, '/', d.day, '/', d.year, '\n');
}

```

Upon running the above program, the following is displayed:

```

7/1/1995
3/24/1994

```

However, like arrays, structures are always passed by reference, regardless of whether or not the **ref** qualifier is used. This means that the called function will always be able to modify the contents of the structure passed in by the parent. For example:

```

#include <max.mh>

struct date
{
    int: year, month, day;
};

void changeit(ref struct date: d)
{
    d.year := 1993;
}

int main()
{
    struct date: d;

    d.year := 1995;
    d.month := 4;
    d.day := 16;

    print(d.month, '/', d.day, '/', d.year, '\n');
    changeit(d);
    print(d.month, '/', d.day, '/', d.year, '\n');

    return 0;
}

```

This sample program will print:

```
4/16/1995
4/16/1993
```

## 11.12. Casts

A *cast* is an explicit instruction to the MEX compiler to convert a data object from one type to another. Casts are not required for most MEX programs, but they are useful in some situations.

A cast is used within an expression like this:

```
( type ) argument
```

*argument* is the expression or object that is to be converted.

*type* is the type to which *argument* is to be converted. MEX currently only allows programs to cast to or from the **char**, **int**, and **long** types.

For example, the following code segment converts a **char** so that it can be assigned to a **long**:

```
char: c;
long: l;

c := 'A';

l := (long)c;           // l now contains 65, the ASCII
                        // value of 'c'.
```

In many cases, casting is not necessary. In the example above, MEX would perform an implicit type conversion anyway if the cast were not specified. The following line would have exactly the same effect:

```
l := c;                 // perform implicit conversion
                        // from char to long.
```

However, casts are sometimes used when calling the **print** function to display output of a certain type. The **print** function is type-sensitive, meaning that it produces different forms of output depending on the type of data object that is passed to it.

For example, if we have the following code:

```
#include <max.mh>
```



```

int main()
{
    char: c;
    int: i;
    long: l;

    c := 65;
    i := 65;
    l := 65;

    print(c, '\n');
    print(i, '\n');
    print(l, '\n');

    return 0;
}

```

The program will display the following output:

```

A
65
65

```

The number 65 is the ASCII value of the letter ‘A’, and since **print** knows that it is displaying a character, the first line in the output is the character representation of *c*.

Likewise, **print** knows when it is displaying integers and long integers, so it formats the output appropriately.

Casting is useful when you have an object of one data type but wish it to be displayed as an object of another type.

For example, suppose that we actually wanted our program to display the ASCII value of *c*. We could then rewrite that one line like this:

```
print( (int)c, '\n' );
```

which would produce the desired output of “65.” The cast converts the character to an integer, so when **print** goes to display the object, it applies the standard integer formatting rules.

Likewise, casts can be useful when dealing with signed and unsigned integers. MEX stores negative numbers using the standard two’s complement notation; this means that “-1” is represented (in a 16-bit **int**) as 65535; “-2” is represented as 65534; and so on.

If your program reads in a value as an unsigned integer but wishes to display the signed representation of that number, the following code can do the job:

```

#include <max.mh>

int main()
{
    unsigned int: ui;

    // Code to read the integer would go here. For this
    // example, we just assign an unsigned value to the
    // 'ui' variable.

    ui := 65532;

    print("Unsigned is ", ui, '\n');
    print("   Signed is ", (int)ui, '\n');

    return 0;
}

```

This code will display:

```

Unsigned is 65532
Signed is -4

```

Likewise, a similar cast can be used to convert a negative signed number into the standard two's-complement (unsigned) representation.

### 11.13. Further Explorations in MEX

This completes the MEX language tutorial. While far from a comprehensive programming design manual, this section should have given you an overview of the capabilities of the MEX language, and it should have also given you some ideas for writing programs of your own.

From here, the best thing to do is to try writing some MEX programs. After you feel comfortable with the basics of the MEX language, also have a look at section 13 later in this document.

In that section, a number of helpful interfacing tips are presented for writing MEX programs that interface with the internal Maximus functions and data structures. Since the goal of writing MEX programs is typically to extend the functionality of Maximus in some way, section 13 is a valuable resource.

Also, section 15 will also prove to be useful for first-time MEX programmers. That section describes how to use all of the functions in the MEX run-time library; you will probably find that many of these functions come in handy when writing your own MEX programs, so it pays to know what is in the run-time library before you start writing.

Good luck in the world of MEX programming!



---

# 12. MEX for C and Pascal Programmers

This section serves as a brief summary of the MEX language for intermediate programmers. A familiarity is assumed with either the C or the Pascal programming language.

This section is very much a “whirlwind tour” of the concepts used in the MEX language. The details given below are probably not enough to allow a C or Pascal developer to start programming in MEX right away.

Instead, this section should be used as a list of key MEX features that are similar to or different from other languages. Much more detailed explanations of these key concepts can be found in section 11.

## 12.1. Comments

A comment is started by the two-character sequence “//” and continues until the end of the line. This concept is borrowed from C++ (and indirectly from BCPL).

## 12.2. Include Files

MEX borrows the “include file” concept from the C language. The **#include** directive instructs MEX to read in the named file and process it as if it were included directly in the source file.

The following line must be present at the top of each MEX file designed to interface with Maximus:

```
#include <max.mh>
```

## 12.3. Blocks

For denoting logical blocks within a function, MEX uses the “{” and “}” characters, as in C. These are equivalent to the “begin” and “end” keywords in Pascal.

## 12.4. Function Definitions

As in C, a subprogram in MEX is always called a function. A “procedure” is equivalent to a function that returns *void*.

A function definition looks like this:

```
returntype name (param-list)
{
    // body goes here
}
```

where *returntype* is the return type of the function, *name* is the name of the function to be defined, and *param-list* is a comma-delimited list of function parameters.

## 12.5. Types

MEX supports the following types: **char**, **int**, **long**, **string**, **struct**, **array**, and **void**. **char**, **int**, and **long** are integral data types. **struct** and **array** are aggregate data types that contain many data objects. **void** is a special data type used only for function return values and **ref void** function parameters.

Both signed and unsigned versions of the three integral types can be declared. Without an explicit **signed** or **unsigned** type qualifier, the **char** type is unsigned, while both **int** and **long** are signed by default.

The **char**, **int**, **long**, **struct** and **void** types are borrowed from C.

The **array** type is borrowed from Pascal.

The **string** type is borrowed from BASIC.

## 12.6. Variable Declarations

A single variable declaration looks like this:

```
type: name-list;
```

*type* can be one of the standard type names, including **char**, **int**, **long**, and **string**. *type* can also be an array type (“array [*lower* .. *upper*] of *type*”) or a structure type (“struct *tag*”). The type name is always followed by a colon (“:”).

*name* is a comma-delimited list describing the names of the variables to be declared. The variable name must abide by these conventions:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that “delta,” “Delta,” “DELTA,” and “DeLtA” refer to four distinct objects.
- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that “top10” is a valid name, whereas “7up” is not.)

The variable declaration syntax is a mix of both C and Pascal declaration styles. Multiple variables declarations can be placed in a block after any opening brace in a function. No keywords are required to define the start or end of a variable declaration block; the compiler recognizes the declarations by context.

A sample function that declares two strings and an integer is given below:

```
void myfunc()  
{  
    string: str1, str2;  
    int: myint;  
  
    // function code goes here  
}
```

## 12.7. Function Prototypes

Before a function may be called, a prototype must be declared that lists the function return value, name, and parameter list. This concept is similar to the C++ “prototype” and the Pascal “forward declaration.” (However, if the function is defined at a point in the same source file above the call to that function, no prototype is required.)

A function prototype looks just like a function definition, except that the “{” and “}” of the function body are replaced by a single semicolon (“;”).

## 12.8. Function Return Values

As in C, values are returned by functions using the *return* keyword. This differs from Pascal where the function return value is set by “assigning” a value to the name of the function.

## 12.9. Strings

MEX supports dynamic strings, as in BASIC. When declaring a string, no explicit string length need be defined. Similarly, strings can be concatenated, assigned, and otherwise manipulated without worrying about the length of the string.

Individual characters can be retrieved from (and inserted into) a string using the “*str* [ *idx* ]” operator, where *str* is a string variable and *idx* is an integer specifying the character number to be modified, where an *idx* of 1 represents the first character in the string.

The length of a string can be determined using the **strlen** function in the MEX run-time library. A number of other string-processing functions can also be found in the run-time library.

## 12.10. Compound Statements

MEX borrows the concept of compound statements from C. At any point in the language where a single statement is accepted, a group of *n* statements (or a *compound statement*) can be specified as shown below:

```
{
    stmt1;
    stmt2;
    // ...
    stmtN;
}
```

A compound statement begins with a left brace and ends with a right brace. Any number of statements can be contained. Compound statements can be nested.

## 12.11. Arithmetic, Relational and Logical Operators

MEX borrows concepts from both C and Pascal in this area.

The assignment and equality operators are borrowed from Pascal. To assign a value to a variable, use the “:=” operator. To test if two variables are equal, use the “=” operator.

The equality operator can operate upon all data types except **array**, **struct** and **void**.

The assignment operator can operate upon all data types except **array** and **void**.



The other arithmetic, relational and logical operators can operate only on the three integral data types, **char**, **int** and **long**. The “+” operator can also be used to concatenate strings.

## 12.12. The for Statement

MEX borrows the syntax for the **for** statement from the C language. The syntax is:

```
for (init-expr; test-expr; post-expr)
    statement
```

*init-expr* is evaluated only once (before any other part of the loop is executed).

*test-expr* is evaluated every time the loop is executed, before the first statement in the loop body is processed. The loop exits when *test-expr* equals zero.

*post-expr* is evaluated after every iteration of the loop.

*statement* is the statement (or compound block) that is performed every time the loop is executed.

## 12.13. Arrays

MEX uses a Pascal-like syntax for declaring arrays:

```
array [lower .. upper] of type: varname;
```

*lower* is the lower bound of the array. This value must be an integer.

*upper* is the upper bound of the array. This value must be an integer.

*type* is the base type used for the data objects within the array.

*varname* is the name of the variable to be declared.

Data objects within an array can be accessed using the standard array operator notation (“[” and “]”). The index value specified within must be of an integral type: either **char**, **int**, or **long**.

## 12.14. Pointers

MEX does not support pointer variables.

## 12.15. Pass-By-Reference Arguments

As in Pascal, arguments passed to a function can be passed by reference. This allows the called function to modify the value of the argument specified by the caller.

To use a pass-by-reference argument, the **ref** keyword is included before the variable type in the function's parameter list, for both the function prototype and the function definition.

## 12.16. Variable-Length Arrays

In a function parameter list, MEX allows the programmer to omit the upper bound for an array declaration. This allows functions to accept variable-length arrays:

```
int process_array(array [1..] of int: my_array,
                  int: size_of_my_array)
{
    // process my_array here
}
```

## 12.17. Structures

MEX uses the C syntax for declaring and defining structures. (Structures are conceptually similar to the Pascal “record” type.) Before using a structure, the structure type must be defined as shown below:

```
struct tag
{
    decl-list
};
```

*tag* is the tag name associated with this structure type.

*decl-list* contains a block of declarations which define the objects contained within the structure. This *decl-list* is identical to a variable declaration block in terms of syntax.

To declare a structure variable, the following syntax is used:

```
struct tag: varname;
```

This declaration instantiates a structure of the previously-defined *tag* type, creating a structure variable with name *varname*.

To access a field within a structure, the structure member operator (“.”) is used. For example, if the structure *foo* contained a field called *my\_string* (of type *string*), a value could be assigned to that field as follows:

```
foo.my_string := "String to go into the foo struct.";
```

## 12.18. Run-Time Library Support

MEX supports an extensive run-time library that performs basic I/O tasks and data manipulation. The run-time library also includes numerous interfaces to internal Maximus functions.



---

# 13. Interfacing MEX with Maximus

This section describes various issues related to interfacing MEX programs with Maximus. While all MEX programs must be run under Maximus, this section concentrates on those programs that tie into special Maximus-specific features and functions.

## 13.1. User Information

Maximus creates a number of important data structures that can be accessed by a MEX program. Of these structures, the most important is the user structure. Maximus creates a global variable called *usr* which contains a copy of information about the current user.

The *usr* variable is accessible to any MEX program that contains the “#include <max.mh>” line at the top of the source file.

The *usr* structure contains everything that Maximus knows about the current user, including the user’s name, privilege level, key settings, date of the user’s last call, and more.

In general, this structure can be read from or written to at will. For example, to display a personal greeting to the user:

```
print("Hello there, ", usr.name, ", how are you?\n");
```

If the user was called Steve Smith, the above line would display:

```
Hello there, Steve Smith, how are you?
```

Similarly, other fields in the user structure can be accessed or displayed:

```
print("You come from ", usr.city, " and you have "  
      "these keys: ", usr.xkeys, '\n');
```

If the user’s city field was set to “Smith Falls,” and if the user had keys 1, 2, and C, the following would be displayed:

```
You come from Smith Falls and you have these keys: 12C
```

Most variables in the user record can also be modified. For example, the following line will give key “D” to the current user and adjust the user’s privilege level to 40:

```
usr.xkeys := usr.xkeys + "D";
usr.priv := 40;
```

Most of the other fields in the user file can be accessed in a similar manner. For example, the *usr.ludate* structure can be used to determine the date of the user’s last call. From there, the **stamp\_to\_long** function could be used to convert that structure into a **long**, which could then be compared with the current date to determine how many seconds had elapsed since the user’s last call.

Section 15 contains a list of all global data structures and their contents, including all of the fields in the user structure.

## 13.2. Message Area Information

Maximus stores information about the current message area in two places: in a structure called *marea*, which contains static information about the area itself, and it also stores data in a secondary structure called *msg*, which contains information about the status of the area and the user’s current message.

As before, these structures are defined in the **max.mh** include file.

The following code is used to display the name of the current message area:

```
print("You are in the ", marea.name, " area.\n");
```

Unlike the user structure, the information in the *marea* structure cannot be changed.

The *msg* structure can also be used to display information about the area that is relevant to the user. For example, to display the user’s current message:

```
print("You just read message #", msg.current, '\n');
```

The *msg* structure also contains information on the number of messages in the area and the highest message number. The *msg.current* field can be adjusted to change the current message number, but the other fields must not be modified.

As before, information on the formats of these structures can be found in section 15.

## 13.3. File Area Information

Similarly, information on the current file area can be found in the *farea* structure. Just like the *marea* structure, it must not be modified by MEX programs.

## 13.4. Changing Message and File Areas

The **msg\_area** and **file\_area** functions display a list of areas and prompt the user to select a new area. The **msgareaselect** and **fileareaselect** functions explicitly move the user to a specific message or file area.

## 13.5. Displaying Output

The standard MEX output function, **print**, is also used for most user-related display tasks. The full set of AVATAR color and cursor controls is available to MEX programs through this function.

Some of the AVATAR sequences, such as simple color-changing commands, have already been predefined in **max.mh**. To use them, simply include the appropriate macro in your program source. For example, to display different parts of the text in white or green:

```
print(COL_WHITE "This text is displayed in white, "
      "but this text " COL_GREEN "here" COL_WHITE
      "is shown in green.\n");
```

If the constant for the color you wish to use is not defined in one of the **COL\_\*** constants in **max.mh**, please see the AVATAR\_ATTR sequence in the table below:

Some of the other predefined AVATAR sequences are shown in Table 13.1:

**Table 13.1 AVATAR Control Sequences**

| Command      | Description   |
|--------------|---|
| AVATAR_ATTR  | Set the current color based on the character following the AVATAR_ATTR sequence. This character is encoded using the standard AVATAR color number, as given in Appendix F. Do not forget to cast this number to a <i>char</i> , as shown in the example following this table. |
| AVATAR_BLINK | The following text will blink. The blinking attribute remains active until the next AVATAR color command.   |
| AVATAR_UP    | Move the cursor up by one line.   |
| AVATAR_DOWN  | Move the cursor down by one line.   |
| AVATAR_LEFT  | Move the cursor left by one column.   |
| AVATAR_RIGHT | Move the cursor right by one column.  |
| AVATAR_CLEOL | Clear from the cursor to the end of the line, using the currently-selected attribute.   |
| AVATAR_GOTO  | Move the cursor to a specified location. The next two characters following the AVATAR_GOTO sequence must indicate the row and column, respectively. (The two following row/column values must be casted to the <b>char</b>  |

|            |  |
|------------|--|
|            | type. Casting is described in section 11.12.)  |
| AVATAR_CLS | Clear the screen, move the cursor to row 1, column 1, and set the current color to cyan. |

---

Based on the user's video setting, these AVATAR codes will be either sent as-is, translated into ANSI codes, or stripped from the display stream, depending on the user's graphics capabilities. This translation is performed automatically by Maximus.

As an example of graphics capabilities, this code clears the screen, moves the cursor to row 5, column 6, sets the color to light green, and then displays the text "Hello."

```
print(AVATAR_CLS AVATAR_GOTO, (char)5, (char)6,
      COL_LGREEN "Hello.\n");
```

To demonstrate the AVATAR\_ATTR sequence, the following code changes the current color to light red on blue. By consulting the AVATAR color table in Appendix F, we see that light red on blue is color 28, so we proceed as follows:

```
print(COL_WHITE "This is a " AVATAR_ATTR, (char)28,
      "gaudy" COL_WHITE " color.\n");
```

## 13.6. Retrieving Input

Maximus supports a number of functions which retrieve input from the user. Of these functions, the most commonly-used are **input\_str**, **input\_list** and **input\_ch**.

These functions use the same interface that Maximus itself uses for retrieving input. These functions can all display prompts and perform special actions depending on what the calling code requires:

The **input\_str** function is used to get either a single word or an entire string from the user.

The **input\_list** function prompts the user to enter a character from a specified list of choices. This is used internally for the "More [Y,n,=]?" prompt and also for most other questions that require single-character responses.

The **input\_ch** function obtains a single character from the user. This function can also interpret the "scan codes" used by terminal programs that support the "Door-Way" protocol.

Also, the **getch** function is used to perform raw character input. It retrieves a character directly from the user, without going through the command stacking buffer or displaying any prompts.



## 13.7. File I/O

For manipulating files on disk, a number of run-time library functions come in handy. **open**, **read**, **write**, **seek**, **tell** and **close** are useful for reading, writing, and updating the contents of files. **fileexists** and **filesize** are used to determine general information about named files. And the **filefind\*** functions can be used to expand wildcards to find files in a directory tree.

## 13.8. Menu Commands, Displaying Files and External Programs

The **menu\_cmd** and **display\_file** functions are used to run internal Maximus menu commands and display **.bbs** files, respectively.

The **shell** function spawns an external program, such as a **.exe**, **.bat**, or **.cmd** file.

## 13.9. Download Queue

The **tag\_queue\_file** function is used to add a file to the download queue. The other **tag\_queue\_\*** functions also allow a MEX program to retrieve information about existing files in the queue.

To send a file to the user, the program must first queue the file using **tag\_queue\_file**. Next, it must call **menu\_cmd** and instruct it to run the internal **MNU\_FILE\_DOWNLOAD** menu option. This will initiate the normal download sequence. Maximus will prompt the user for a protocol if necessary, and it will also allow the user to select other files to download.

If this behavior is not desired, the MEX program can:

- Set the *usr.def\_proto* field to one of the *PROTOCOL\_\** constants in **max.mh**. This causes Maximus to skip the “Select protocol:” prompt and forces the user to download using a specific protocol.
- Insert a “|” into the *input* global variable. This adds a carriage return to the stacked input string, so the “File(s) to download (#2):” prompt will be skipped and the file will be sent immediately.

## 13.10. Other Functions

While this section has touched on the major functions that are used to interface with Maximus, many of the other functions in the run-time library can be used to modify

## **234     13. Interfacing MEX with Maximus**

Maximus's internal behavior. Please consult the function list in section 15 for more information.

---

# 14. MEX Compiler Reference

## 14.1. Command Line Format

The format for invoking the MEX compiler is given below:

```
MEX filename [-a] [-hsize] [-s] [-q]
```

**filename** is the name of the MEX source file to compile. If no file extension is specified, **.mex** is assumed.

The command line switches supported by MEX are shown below in Table 14.1:

**Table 14.1 MEX Command Line Switches**

| Parameter | Description  |
|-----------|--|
| -a        | Display addresses instead of symbolic names when writing quad output. This parameter is only valid when used in conjunction with the -q parameter.   |
| -h<size>  | Set the program heap size to <size>. The heap is used for storing dynamic strings. The default heap size is 8192 bytes, but this limit may need to be increased if a large number of strings are used within one MEX program.  |
| -s<size>  | Set the program stack size to <size>. The stack is used for storing local variables and for performing recursive function calls. The default stack size is 2048 bytes, but this limit may need to be increased if a program uses many local variables or makes a large number of recursive function calls.   |
| -q        | Instead of writing out a <b>.vm</b> file as p-code, write the output to the console as a sequence of ASCII quadruples (quads). This is equivalent to the “assembler output” option of most native-code compilers. A quad is a 4-tuple consisting of an operator, two input arguments, and a destination. The quad listing can sometimes be used to help debug a program. |

The MEX compiler converts the MEX program source into *compiled p-code* which is interpreted at run-time by Maximus. Generating p-code has two main benefits over generating native object code:

- p-code is not operating system or processor-specific, so the same MEX program can run under either 16-bit DOS or 32-bit OS/2 environments without recompilation.
- Although executing p-code is somewhat slower than executing native object code, over 120 functions are included in the standard MEX run-time library that can be used to access internal Maximus features, perform input and output, manipulate strings, and manage information. Since these library functions are embedded within Maximus, they run at native object code speeds. The more a MEX program takes advantage of the run-time library, the faster it runs.

## 14.2. Environment Variables

When searching for files specified in **#include** directives, the MEX compiler will first look for the include file relative to the current directory. If the file cannot be found there, MEX will examine the *MEX\_INCLUDE* environment variable. This variable must contain a semicolon-delimited list of paths where MEX include files can be found. Maximus will then search all of the directories listed in this environment variable.

For example, by entering this at the command prompt:

```
SET MEX_INCLUDE=d:\max\m;e:\mextest
```

when MEX encounters an **#include** directive, if the file cannot be found in the current directory, it will look for the include file in the **d:\max\m** and **e:\mextest** directories.

## 14.3. Error Messages and Warnings

The following error messages and warnings are generated by the MEX compiler:

### 14.3.1. General Syntax Errors

2000 syntax error near *symbol*

MEX could not make any sense of the source code that is near the *symbol* token. Check your code for misplaced semicolons or other common errors.

2001 invalid identifier type

You attempted to use a function or structure name as a variable.

**14.3.2. Preprocessor Errors**

- 2100 can't open file *symbol* for read
- MEX could not find the named file. Ensure that the file exists and that your MEX\_INCLUDE environment variable is set correctly.
- 2101 invalid #include directive
- MEX could not make any sense of an include directive. Ensure that a filename is specified.
- 2102 invalid #include file
- MEX could not make any sense of an include filename. Ensure that the filename is surrounded by angle brackets.
- 2103 too many nested include files
- MEX can only handle up to 16 nested include files at a time. Reduce the number of nested **#include** directives.
- 2104 invalid #ifdef/#ifndef
- The **#ifdef** or **#ifndef** statement is invalid. Ensure that a proper constant or define follows the **#ifdef** or **#ifndef** directive.
- 2105 too many nested #ifdef/#ifndef
- MEX only supports up to 8 nested **#ifdef** or **#ifndef** statements.
- 2106 unmatched #ifdef directive
- A matching **#endif** could not be found for an **#ifdef** directive.
- 2107 unmatched #endif directive
- A matching **#ifdef** could not be found for this **#endif** directive.
- 2108 unmatched #else directive
- A matching **#ifdef** could not be found for this **#else** directive.
- 2109 unmatched #elifdef
- A matching **#ifdef** could not be found for this **#elifdef** directive.

2110 invalid #define

This **#define** directive has an incorrect format. Ensure that it includes a variable to be defined.

2111 macro *symbol* is already defined and is not identical

The *symbol* macro was defined using a previous **#define** statement, but the current **#define** directive attempts to change the value of *symbol* to something else.

2112 #error directive: *string*

An **#error** directive was encountered in the program source.

2113 unknown directive *symbol*

An unknown preprocessor directive was encountered in the source file.

#### 14.3.3. Lexical Errors

2200 unterminated character constant

A character constant was specified, but it included a newline or end-of-file instead of a character.

2201 unterminated string constant

A string constant was specified, but it did not include a closing double quote.

2202 invalid character constant

A character constant was specified, but it did not include a closing single quote.

2203 invalid hex escape sequence “\x*symbol*”

The specified hexadecimal escape sequence is not a valid hexadecimal number.

2204 invalid character *symbol*

An invalid character was found in the source file, such as an unrecognized punctuation symbol or a high-bit character.

**14.3.4. Type Mismatch Errors**

- 2300 type must not be 'void'
- The **void** type can only be used in certain locations, such as in function return values and in "void ref" function parameters.
- 2301 symbol *symbol* is not a structure type
- You attempted to use *symbol* as a structure tag, even though no structure type definition for that tag was found.
- 2302 invalid type *symbol* for *type* statement
- The argument for a **if**, **while**, **for**, or **do .. while** expression was not correct. Only **char**, **int**, and **long** can be used in these expressions.
- 2303 can't use function as a variable
- You attempted to use a function name in a context that required a variable name.
- 2304 the type 'void' cannot have a value
- You attempted to manipulate a function returning **void** as part of an expression.
- 2305 invalid type conversion: *type1* -> *type2*
- It is impossible to convert an object of *type1* to *type2*. This is usually the result of incompatible types for explicit casts, passing parameters to functions, and passing back function return values.
- 2306 *symbol* must be an array
- You attempted to use the array operator ("[" and "]") on the variable *symbol* which is not an array.
- 2107 can't use [] on a non-array
- An expression on the specified line cannot be used with the array operator.
- 2308 invalid index type for array *symbol*
- The index type used to reference an array must be integral: either **char**, **int**, or **long**. The array *symbol* cannot be indexed using any other types.

## 240 14. MEX Compiler Reference

2309 out-of-range subscript (*num*) for array *symbol*

You attempted to access a subscript that was outside of the bounds for the array *symbol*.

2310 dot operator can only be used with structures

You attempted to use the dot operator (“.”) with a variable that was not a structure.

2311 *symbol* is not a member of struct *name*

The field *symbol* was not found in the structure definition for the structure with a tag name of *name*.

2312 *symbol* not a goto label

The target of a **goto** must be a label, not a variable or a function name.

2313 lvalue required

The target of an assignment must be a simple variable or the result of the array or structure operators. You cannot assign a value to an expression. (For example, “*i*+2 := 2” is invalid, since “*i*+2” is an expression and cannot be assigned a value.)

2314 cannot apply unary minus to *symbol*

The unary minus operator cannot be applied to the *symbol* object. The unary minus can only be applied to the **char**, **int**, and **long** types.

2315 declared array must have upper bound

An array declared in a function body must have an upper bound. Only arrays declared as function parameters can omit the upper bound.

2316 cannot apply sizeof() to a boundless array

The **sizeof** operator can only be used on arrays of finite dimensions.

### 14.3.5. Symbol Table Errors

2400 redeclaration of *symbol*

The variable or function *symbol* has already been defined in this scope.



- 2401 redeclaration of structure *symbol*
- The structure type *symbol* has already been declared in this scope.
- 2402 undefined structure name: *symbol*
- The structure tag *symbol* is unknown.
- 2403 symbol *symbol* is not a structure type
- The tag *symbol* specified for a structure was the name of another variable or function, rather than a proper structure tag.
- 2404 undefined label *symbol*
- The label *symbol* was not found anywhere in the function in which it was used.
- 2405 undefined variable *symbol*
- The variable *symbol* has not been declared.
- 2406 struct *symbol* must be declared before use
- Your program attempted to use a structure type that has not yet been defined.
- 2407 invalid redeclaration of function *symbol*
- A function was declared using the same name as a global variable or structure tag.
- 2408 redeclaration of argument *symbol*
- The function argument *symbol* has been declared twice in the parameter list.
- 2409 redeclaration of function body for *symbol*
- The function body for *symbol* has already been encountered in the source file. A function can only be defined once.
- 2410 invalid range: '*num* .. *num*'
- The lower bound for an array was greater than the upper bound.

**14.3.6. Function-Related Errors**

- 2501 argument mismatch: function=*type*, prototype=*type*
- The type of parameter specified in the function's argument list differs from that specified in the function prototype.
- 2502 too many arguments in function declaration
- Too many arguments were specified in the function declaration. The function prototype specified a fewer number of arguments.
- 2503 too few arguments in function declaration
- Too few arguments were specified in the function declaration. The function prototype specifies a larger number of arguments.
- 2504 call to *symbol* with no prototype
- A function was called, but the function was neither defined nor prototype earlier in the source file.
- 2505 variable *symbol* is not a function
- You attempted to make a call to *symbol*, even though it is a variable and not a function.
- 2506 lvalue required (arg *num* of *symbol*)
- In a call to function *symbol*, argument number *num* is pass-by-reference. This means that a variable of the appropriate type (and not an expression or constant) must be passed for the specified argument.
- 2507 not enough arguments in call to *symbol*
- Not enough arguments were specified in the call to *symbol*, relative to the function prototype.
- 2508 too many arguments in call to *symbol*
- Too many arguments were specified in the call to *symbol*, relative to the function prototype.
- 2509 void function cannot return a value
- You attempted to return a value from a function that was declared as returning *void*.

2510 non-void function must return a value

You used a simple “return;” statement with no value, even though the function itself must return a value.

#### 14.3.7. Warnings

3000 constant truncated: *symbol*

The numeric constant *symbol* was too long and had to be truncated.

3001 identifier truncated: *symbol*

The identifier *symbol* was too long and had to be truncated at 32 characters.

3002 meaningless use of an expression

You attempted to use the equality operator in a way that does not make sense. For example, the following line of code:

```
i = 2;
```

will produce this warning because you are simply comparing the value of *i* to 2, but then throwing away the result of the comparison. You probably meant to write:

```
i := 2;
```

which assigns the value 2 to *i*.



---

# 15. MEX Library Reference

This section describes all of the functions and structures available in the standard MEX run-time library.

## 15.1. Global Variables and Data Structures

Maximus exports certain global variables to all MEX programs. These variables can be used to track the internal system state, including information about the current message and file area, screen settings, user information, and more.

All of the following global variables are declared in the **max.mh** include file. The following line must be present at the top of a MEX source file to access these variables:

```
#include <max.mh>
```

### 15.1.1. Strings Exported by Maximus

This section describes the strings that are exported by Maximus to all MEX programs.

---

#### input

---

|             |  |
|-------------|--|
| Declaration | string: input;   |
| Description | <p>This string contains the current “stacked keyboard input string” used by Maximus. The <i>input</i> string contains a sequence of keys that will be used to try to satisfy the input requirements of following calls to the <b>input_ch</b>, <b>input_str</b>, and <b>input_list</b> functions. The internal Maximus input commands also use this buffer for their own input.</p> <p>The internal Maximus input functions also try to take their input from this string before prompting the user for input.</p> |

The **input\_\*** functions will draw as many characters as necessary from this string to satisfy an input request. (In the case of **input\_ch** or **input\_list**, only a single character will normally be removed. In the case of **input\_str**, a single word or an entire line is normally removed.)

This string can be modified by MEX programs.

### 15.1.2. Structures Exported by Maximus

This section describes the structures that are exported by Maximus to all MEX programs:

---

---

#### struct \_instancedata

---

---

#### Declaration

```
struct _instancedata: id;
```

#### Description

This structure contains information about the current Maximus session. The contents of the structure are shown below in Table 15.1:

**Table 15.1** struct \_instancedata

| Field         | Type         | Description   |
|---------------|--------------|---|
| instant_video | int          | <p>This controls the “instant video” setting. If instant video is enabled, output displayed by <b>print</b> will be shown on the local screen as soon as the <b>print</b> function is called.</p> <p>However, screen updating can be a slow process, so it is sometimes more efficient to disable local screen writes, perform many <b>print</b> calls, and to then display the local screen again (using the <b>vidsync</b> function) after all of the <b>print</b> calls have been completed.</p> <p>To disable the instant screen updating feature, set <i>id.instant_video</i> to 0. While this variable is set to 0, the <b>print</b> function will not cause the local screen to be updated.</p> <p>However, be warned that some internal Maximus functions will update the screen regardless of this setting. Examples of this are <b>input_ch</b>, <b>input_str</b> and <b>input_list</b>.</p> <p>This field can be modified by MEX programs.</p> |
| task_num      | unsigned int | The current task number.  |
| local         | int          | TRUE if the caller is logged on at the local console.   |

|              |               |   |
|--------------|---------------|---|
| port         | int           | Under DOS: the current COM port number.<br>Under OS/2: the current communications handle. |
| speed        | unsigned long | The user's connection speed, in bits per second (bps).                                    |
| alias_system | int           | TRUE if the system supports aliases ("Alias System" in the system control file).          |
| ask_name     | int           | TRUE if the system will ask the user for a real name.                                     |
| use_umsgid   | int           | TRUE if the system displays UMSGIDs instead of message numbers.                           |

---

## struct \_marea

---

**Declaration**

```
struct _marea: marea;
```

**Description**

This structure contains information about the current message area. Structures of this type are also returned by the **msgareafind\*** functions. The contents of the structure are shown below in Table 15.2:

**Table 15.2 struct \_marea**

| Field      | Type   | Description   |
|------------|--------|---|
| name       | string | This name of the message area.  |
| descript   | string | The description for the message area.   |
| path       | string | The path and/or filename of the message area. (*.MSG areas will be provided as a path; Squish areas will be provided as a path and filename.) |
| tag        | string | The tag for this area (for use in EchoMail applications).   |
| attachpath | string | The path for local file attaches.   |
| barricade  | string | The barricade filename for this area.   |
| division   | int    | Non-zero if this is a message division starting or ending record. (1=beginning; 2=ending.)  |
| type       | int    | Message area type. One of either MSGTYPE_SQUISH or MSGTYPE_SDM (*.MSG).   |
| attribs    | int    | Message area attributes. See the MA_* definitions in <b>max.mh</b> .  |

---

## struct \_farea

---

**Declaration**

```
struct _farea: farea;
```

**Description**

This structure contains information about the current file area. Structures of this type are also returned by the **fileareafind\*** functions. The contents of this structure are shown in Table 15.3:

**Table 15.3** struct \_farea

| Field     | Type   | Description   |
|-----------|--------|---|
| name      | string | This name of the file area.   |
| descript  | string | The description for the file area.  |
| downpath  | string | The download path for this file area.   |
| uppath    | string | The upload path for this file area.   |
| filesbbs  | string | Path to the <b>files.bbs</b> -like list for this file area.                             |
| barricade | string | The barricade filename for this area.   |
| division  | int    | Non-zero if this is a file division starting or ending record. (1=beginning; 2=ending.) |
| attribs   | int    | File area attributes. See the FA_* definitions in max.mh.                               |

---

## struct \_msg

---

**Declaration**

```
struct _msg: msg;
```

**Description**

This structure contains the current message number and other related information. The contents of the structure are shown below in Table 15.4:

**Table 15.4** struct \_msg

| Field     | Type | Description  |
|-----------|------|--|
| current   | long | The current message number, or if <b>Use UMSGIDs</b> is enabled, the UMSGID of the current message number. This field can be modified by MEX programs. |
| high      | long | The highest message number in the current area, or if <b>Use UMSGIDs</b> is enabled, the UMSGID of the highest message number.                         |
| num       | long | The number of messages in the current area.  |
| direction | int  | Message reading direction. 1 = next; 0 = prior.  |

---

## struct \_usr

---

**Declaration**

```
struct _usr: usr;
```

**Description**

This structure contains information about the current user. Structures of this type are also manipulated by the **user\*** functions.

Table 15.5 describes the contents of this structure. All fields in this structure can be modified by MEX programs.



Table 15.5 struct \_usr

| Field        | Type         | Description   |
|--------------|--------------|---|
| name         | string       | The user's name.  |
| city         | string       | The user's city and state/province.   |
| alias        | string       | The user's alias.   |
| phone        | string       | The user's voice telephone number   |
| lastread_ptr | unsigned int | An integer that is unique among all users in the system. This number is used as an index for the lastread pointers in the Squish SQI files, to find records in the \max\olr\dats directory, and in numerous other places. |
| pwd          | string       | The user's password. If the "usr.encrypted" flag is set, this may not be a readable password.   |
| times        | unsigned int | The number of previous calls to the system made by this user.   |
| help         | char         | The user's help level. (Novice = 6; Regular = 4; Expert = 2.)   |
| video        | char         | The user's video mode. (0 = TTY; 1 = ANSI; 2 = AVATAR.)   |
| nulls        | char         | The number of NUL characters sent after a carriage return.  |
| hotkeys      | char         | TRUE if hotkeys are enabled.  |
| notavail     | char         | TRUE if the user is not available for multi-node chatting.  |
| fsr          | char         | TRUE if the user has enabled the full-screen reader.  |
| nerd         | char         | TRUE if the user is a nerd (meaning that the user's yells are silenced).  |
| noulist      | char         | TRUE if the user does not show up in the userlist.  |
| tabs         | char         | TRUE if the user's terminal can handle tab characters.  |
| encrypted    | char         | TRUE if the user's password is encrypted.   |
| rip          | char         | TRUE if the user has enabled RIPscrip graphics.   |
| badlogon     | char         | TRUE if the user's last log-on attempt resulted in a failed password.   |
| ibmchars     | char         | TRUE if the user has enabled the high-bit IBM character set.  |
| bored        | char         | TRUE if the line editor is enabled; otherwise, the full-screen MaxEd is used.   |
| more         | char         | TRUE if the "More" prompt is enabled.   |
| configured   | char         | TRUE if the user's city and password fields have been set.  |
| cls          | char         | TRUE if the user's terminal can handle  |

|             |               |  |
|-------------|---------------|--|
|             |               | screen clears.   |
| priv        | unsigned int  | The user's privilege level.  |
| dataphone   | string        | The user's data phone number.  |
| time        | unsigned int  | The length of time (in minutes) that the user has been on-line today.  |
| deleted     | char          | TRUE if the user has been deleted.   |
| permanent   | char          | TRUE if the user is undeletable.   |
| msgs_posted | long          | The total number of messages posted by this user.  |
| msgs_read   | long          | The total number of messages read by this user.  |
| width       | char          | The width of the caller's screen. Also see the <code>term_width</code> and <code>screen_width</code> functions in the run-time library reference.  |
| len         | char          | The height of the caller's screen. Also see the <code>term_length</code> and <code>screen_length</code> functions in the run-time library reference.   |
| credit      | unsigned int  | User's NetMail credit, in cents.   |
| debit       | unsigned int  | User's NetMail debit, in cents.  |
| xp_priv     | unsigned int  | Priv level to which the user will be demoted when the current subscription expires.  |
| xp_date     | struct _stamp | Date and time for the user's subscription expiry.  |
| xp_mins     | unsigned long | Number of minutes remaining in the user's subscription.  |
| expdate     | char          | TRUE if the user expires based on the <code>xp_date</code> field.  |
| expmins     | char          | TRUE if the user expires based on the <code>xp_mins</code> field.  |
| expdemote   | char          | TRUE if the user is to be demoted to the value in the <code>xp_priv</code> field when the subscription expires.  |
| expaxe      | char          | TRUE if Maximus should hang up on the caller when the subscription expires.  |
| sex         | char          | The user's gender: either <code>SEX_MALE</code> , <code>SEX_FEMALE</code> or <code>SEX_UNKNOWN</code> .  |
| ludate      | struct _stamp | Date of the user's last call.  |
| xkeys       | string        | The user's keys (as a string).   |
| lang        | char          | The user's current language number. The <code>language_num_to_string</code> function is to translate this number into a string.  |
| def_proto   | char          | The user's default protocol. This can be one of the <code>PROTOCOL_*</code> defines from <b>max.mh</b> , or it can be a positive index to indicate an external protocol. The <code>protocol_num_to_string</code> function is used to translate all of these numbers (for both internal |

|              |               |  |
|--------------|---------------|--|
| up           | unsigned long | and external protocols) into strings.<br>Total kilobytes uploaded for all calls.   |
| down         | unsigned long | Total kilobytes downloaded for all calls.  |
| downtoday    | unsigned long | Total kilobytes downloaded today.  |
| msg          | string        | The user's current message area.   |
| files        | string        | The user's current file area.  |
| compress     | char          | The user's default compression program.<br>The compressor_num_to_string function is used to translate this number into a string. |
| dob          | string        | The caller's date of birth (in "yyyy.mm.dd" format).   |
| date_1stcall | struct _stamp | Date/time of the user's first call to the system.  |
| date_pwd_chg | struct _stamp | Date/time of the user's most recent password change.   |
| nup          | unsigned long | Total number of files uploaded to the system.  |
| ndown        | unsigned long | Total number of files downloaded from the system.  |
| ndowntoday   | unsigned long | Number of files downloaded from the system today.  |
| time_added   | unsigned int  | Credits added to the user's time today (for chatting with the SysOp or for upload time credits).                                 |
| point_credit | unsigned long | Total number of point credits.   |
| point_debit  | unsigned long | Total number of point debits.  |
| date_newfile | struct _stamp | Date/time of last "new files" search.  |
| call         | unsigned int  | Number of previous calls today.  |

---

## struct \_sys

---

**Declaration**

```
struct _sys: sys;
```

**Description**

This structure contains information about the current system screen display. The contents of this structure are described in Table 15.6:

**Table 15.6 struct \_sys**

| Field       | Type | Description   |
|-------------|------|---|
| current_row | int  | The current cursor row position.                                    |
| current_col | int  | The current cursor column position.                                 |
| more_lines  | int  | The number of lines displayed since the last "More [Y,n,=]" prompt. |

**15.1.3. Other Data Structures**

The structures listed in this section are not exported directly by Maximus. However, the structures are used (directly or indirectly) by some of the functions in the run-time library. Please consult section 15.3 for more information on the structures used by a particular function.

---

**struct \_date**


---

**Description**

This structure is used in various places to represent a system date. The contents of the structure are described in Table 15.7:

**Table 15.7 struct \_date**

| Field | Type | Description                                |
|-------|------|--|
| day   | char | The day of the month. (1 = the first day.) |
| month | char | The month of the year. (1 = January.)      |
| year  | char | The current year less 1980. (0 = 1980.)    |

---

**struct \_time**


---

**Description**

This function is used in various places to represent a system time. The contents of this structure are described in Table 15.8:

**Table 15.8 struct \_time**

| Field | Type | Description  |
|-------|------|--|
| hh    | char | Hour in 24-hour format. (0 = midnight; 12 = noon, 17 = 5 P.M.) |
| mm    | char | Minute. (Must be within 0 - 59 inclusive.)                     |
| ss    | char | Second. (Must be within 0 - 59 inclusive.)                     |

---

**struct \_stamp**


---

**Description**

This structure contains copies of both a date and time structure to provide a complete system timestamp. The contents of this structure are described in Table 15.9:

**Table 15.9 struct \_stamp**

| Field | Type         | Description                   |
|-------|--------------|-------------------------------|
| date  | struct _date | A copy of the date structure. |
| time  | struct _time | A copy of the time structure. |

---

## struct \_cstat

---

**Description**

This structure is used by the **chat\_querystatus** function. The contents of this structure are described in Table 15.10:

**Table 15.10 struct \_cstat**

| Field    | Type   | Description   |
|----------|--------|---|
| task_num | int    | The user's task number                                  |
| avail    | int    | TRUE if the user is available for chat                  |
| username | string | The user's name   |
| status   | string | A status message describing the user's current actions. |

---

## struct \_ffind

---

**Description**

This structure is used by the **filefind\*** functions to retrieve file status information. The contents of this structure are described in Table 15.11:

**Table 15.11 struct \_ffind**

| Field    | Type          | Description   |
|----------|---------------|---|
| finddata | long          | Instance-specific find data. This information is internal to Maximus and must not be modified by the MEX program. |
| filename | string        | The name of the file.   |
| filesize | unsigned long | The size of the file.   |
| filedate | struct _stamp | The file's modification date/time.  |
| fileattr | unsigned int  | The file's attributes. See the FA_* definitions in max.mh.  |

---

## struct \_callinfo

---

**Description**

This function is used by the **call\*** functions to read records from the caller log file. The contents of this structure are described in Table 15.12:

**Table 15.12 struct \_callinfo**

| Field        | Type          | Description   |
|--------------|---------------|---|
| name         | string        | The user's name.  |
| city         | string        | The user's city.  |
| login        | struct _stamp | The user's log-on time.   |
| logoff       | struct _stamp | The user's log-off time.  |
| task         | int           | The Maximus task number that created this record.   |
| flags        | int           | A combination of the CALL_* definitions from max.mh.  |
| logon_priv   | unsigned int  | The user's privilege level at log-on.   |
| logon_xkeys  | string        | The user's key settings at log-on.  |
| logoff_priv  | unsigned int  | The user's privilege level at log-off.  |
| logoff_xkeys | string        | The user's key settings at log-off.   |
| filesup      | unsigned int  | Number of files uploaded in this session.   |
| filedn       | unsigned int  | Number of files downloaded in this session.   |
| kbusp        | unsigned int  | Number of kilobytes uploaded in this session.   |
| kbdn         | unsigned int  | Number of kilobytes downloaded in this session.   |
| calls        | unsigned int  | Number of calls user made to the system (as of when the record was written).  |
| read         | unsigned int  | Number of messages read during the session.   |
| posted       | unsigned int  | Number of messages posted during the session.   |
| paged        | unsigned int  | Number of times that the SysOp was paged during the session.  |
| added        | int           | Number of minutes that were credited to the user's time for the day due to file uploads or chatting with the SysOp. |

## 15.2. Functions by Category

This section gives a categorized listing of functions in the MEX run-time library. The functions are first listed by category; however, the next section gives a detailed description of each function, including the function prototype, arguments, return value, and overall function behavior.

### 15.2.1. Screen Output and Display Formatting

The functions listed in Table 15.13 are used for printing text, displaying output, and getting/setting screen formatting information.

**Table 15.13 Screen Output and Display Formatting Functions**

| Function | Description                                |
|----------|--|
| do_more  | Optionally displays a "More [Y,n]" prompt. |

|               |  |
|---------------|--|
| issnoop       | Determines if “Snoop Mode” is active.                            |
| print         | Sends text to the user.  |
| reset_more    | Resets the counter for a “More” prompt.                          |
| screen_length | Returns the length of the system console.                        |
| screen_width  | Returns the width of the system console.                         |
| set_output    | Enables or disables local/remote output.                         |
| set_textsize  | Sets the size of the text window (for <i>RIPscrip</i> graphics). |
| snoop         | Enables or disables “Snoop Mode.”                                |
| term_length   | Returns the length of the user’s virtual terminal.               |
| term_width    | Returns the width of the user’s virtual terminal.                |
| vidsync       | Updates the local video screen after a call to print.            |

### 15.2.2. Keyboard Input

The functions listed in Table 15.14 are used for getting input from the user and for processing keystrokes.

**Table 15.14 Keyboard Input Functions**

| Function   | Description  |
|------------|--|
| getch      | Returns the next raw keystroke entered by the user.            |
| input_ch   | Gets a character from the user (with formatting).              |
| input_list | Gets a character from the user (from a list of choices).       |
| input_str  | Gets an entire word or string from the user.                   |
| iskeyboard | Determines if local keyboard mode is enabled.                  |
| kbhit      | Checks to see if the user has pressed a key.                   |
| keyboard   | Enables/disables local keyboard mode.                          |
| localkey   | Determines if the last keystroke was entered by a remote user. |

### 15.2.3. External Programs, .BBS files, and Menu Options

The functions listed in Table 15.15 are used to interface to non-MEX scripts or programs, such as *.bbs* files, Maximus menu commands, and external programs.

**Table 15.15 External Program Functions**

| Function     | Description  |
|--------------|--|
| display_file | Displays a .BBS, .GBS or .RBS file.                    |
| menu_cmd     | Runs most internal Maximus menu commands.              |
| shell        | Shells to an external program (.exe, .bat, .cmd, etc.) |

### 15.2.4. File I/O and Disk Operations

The functions listed in Table 15.16 support opening, reading, writing, and performing other types of file manipulation and information-gathering.

**Table 15.16 File and Disk Operations**

| Function   | Description  |
|------------|--|
| close      | Closes a file handle.                                |
| filecopy   | Copies a file from one location to another.          |
| filedate   | Retrieves the date for a file.                       |
| fileexists | Determines if a file exists.                         |
| filesize   | Returns the size of a file.                          |
| open       | Opens a file for read or write.                      |
| read       | Reads a block from a file.                           |
| readln     | Reads an entire line from a file.                    |
| remove     | Deletes a file.                                      |
| rename     | Renames a file.                                      |
| seek       | Sets a file's "next read/write location" pointer.    |
| tell       | Returns a file's "next read/write location" pointer. |
| write      | Write a block to a file.                             |
| writeln    | Write an entire line to a file.                      |

**15.2.5. File Searching Operations**

The functions listed in Table 15.17 are used to expand filename wildcards and/or perform file directory searches.

**Table 15.17 File Searching Operations**

| Format        | Description                                      |
|---------------|--|
| filefindclose | Terminates a file-find search.                   |
| filefindfirst | Starts looking for a named file (or a wildcard). |
| filefindnext  | Find the next file after starting a search.      |

**15.2.6. Strings**

The functions listed in Table 15.18 are used for common string manipulation operations, such as padding, stripping, finding substrings, and more.

**Table 15.18 String Operations**

| Function   | Description  |
|------------|--|
| strfind    | Finds the index of a substring within another string.  |
| stridx     | Finds the index of a character within a string.  |
| strlen     | Returns the length of a string.  |
| strlower   | Converts a string to lowercase.  |
| strpad     | Right-pads a string with a certain character.  |
| strpadleft | Left-pads a string with a certain character.   |
| stridx     | Returns the index of the rightmost character within a string that contains a specific value. |



|          |   |
|----------|---|
| strtok   | Tokenizes a string.   |
| strupper | Converts a string to uppercase.   |
| strtrim  | Remove unwanted characters on either end of a string.                       |
| substr   | Extracts a substring from another, given the substring's size and location. |

### 15.2.7. Time and Date

The functions listed in Table 15.19 are used to check the current system time, in addition to setting and querying information about the user's current time limit.

**Table 15.19 Time and Date Functions**

| Function       | Description   |
|----------------|---|
| long_to_stamp  | Converts a long-format timestamp into a string.               |
| stamp_string   | Converts a stamp structure into a string.                     |
| stamp_to_long  | Converts a stamp structure into a long.                       |
| time           | Returns the current time as a long.                           |
| time_check     | Check the user's current time limit.                          |
| timeadjust     | Adjust a user's time limit.                                   |
| timeadjustsoft | Adjust a user's time limit without overrunning events.        |
| timeleft       | Returns the length of time remaining in this session.         |
| timeon         | Returns the amount of time that the user has been on-line.    |
| timestamp      | Returns the current time as a stamp structure.                |
| xfertime       | Calculates the approximate time required for a file transfer. |

### 15.2.8. Time Delay

The function listed in Table 15.20 is used for pausing program operation.

**Table 15.20 Time Delay Functions**

| Function | Description  |
|----------|--|
| sleep    | Causes the system to delay for a certain period of time. |

### 15.2.9. Type Conversions

The functions listed in Table 15.21 are used to convert data between the *string* type and one of the integral types.

**Table 15.21 Type Conversion Functions**

| Function | Description                      |
|----------|----------------------------------|
| itostr   | Converts an integer to a string. |
| ltostr   | Converts a long to a string.     |
| strtoi   | Converts a string to an integer. |
| strtol   | Converts a string to a long.     |

|         |   |
|---------|---|
| uitostr | Converts an unsigned integer to a string. |
| ultostr | Converts an unsigned long to a string.    |

### 15.2.10. Modem and Communications

The functions listed in Table 15.22 are used to directly control the modem.

**Table 15.22 Modem and Communications Functions**

| Function    | Description                                 |
|-------------|---|
| carrier     | Checks whether or not DCD is present.       |
| dcd_check   | Enables or disables automatic DCD checking. |
| mdm_command | Sends a command to the modem.               |
| mdm_flow    | Enables or disables flow control.           |

### 15.2.11. Message Areas

The functions listed in Table 15.23 are used to select the current message area and to search the message area data file.

**Table 15.23 Message Area Functions**

| Function         | Description  |
|------------------|--|
| msg_area         | Displays the message-area menu.                            |
| msgareafindclose | Terminates a message-area finding session.                 |
| msgareafindfirst | Starts a message-area finding session.                     |
| msgareafindnext  | Finds the next area in a message-area finding session.     |
| msgareafindprev  | Finds the previous area in a message-area finding session. |
| msgareaselect    | Sets the user's current message area.                      |

### 15.2.12. File Areas

The functions listed in Table 15.24 are used to select the current file area and to search the file area data file.

**Table 15.24 File Area Functions**

| Function          | Description   |
|-------------------|---|
| file_area         | Displays the file-area menu.                            |
| fileareafindclose | Terminates a file-area finding session.                 |
| fileareafindfirst | Starts a file-area finding session.                     |
| fileareafindnext  | Finds the next area in a file-area finding session.     |
| fileareafindprev  | Finds the previous area in a file-area finding session. |
| fileareaselect    | Sets the user's current file area.                      |

**15.2.13. File Tag Queue**

The functions listed in Table 15.25 are used to manipulate the queue of files to be downloaded.

**Table 15.25 File Tag Queue Functions**

| Function         | Description                                      |
|------------------|--|
| tag_dequeue_file | Removes a file from the tag queue.               |
| tag_get_name     | Retrieves information about a file in the queue. |
| tag_queue_file   | Inserts a file in the queue.                     |
| tag_queue_size   | Returns the number of files in the queue.        |

**15.2.14. User File**

The functions listed in Table 15.26 are used to expand strings in user records, modify records in the user file, and to search the user file for selected records.

**Table 15.26 User File Functions**

| Function               | Description  |
|------------------------|--|
| compressor_num_to_name | Converts the “usr.compress” field to a string.                 |
| language_num_to_name   | Converts the “usr.lang” field to a string.                     |
| protocol_num_to_name   | Converts the “usr.def_proto” field to a string.                |
| usercreate             | Creates a new user record.                                     |
| userfilesize           | Returns the size of the user file.                             |
| userremove             | Deletes a user record.   |
| userupdate             | Updates an existing user record.                               |
| userfindclose          | Terminates a user-file finding session.                        |
| userfindnext           | Finds the next user record in a user-file finding session.     |
| userfindopen           | Starts a user-file finding session.                            |
| userfindprev           | Finds the previous user record in a user-file finding session. |
| userfindseek           | Seeks to a specific user in a user-file finding session.       |

**15.2.15. Caller File Manipulation**

The functions listed in Table 15.27 are used to read the contents of the caller log file.

**Table 15.27 Caller File Functions**

| Function     | Description   |
|--------------|---|
| call_close   | Closes the caller log file.                           |
| call_numrecs | Returns the number of records in the caller log file. |
| call_open    | Opens the caller log file.                            |

|                        |                                 |
|------------------------|---------------------------------|
| <code>call_read</code> | Reads from the caller log file. |
|------------------------|---------------------------------|

**15.2.16. Log File**

The functions listed in Table 15.28 are used to append to the system log file.

**Table 15.28 Log File Functions**

| Function         | Description                    |
|------------------|--------------------------------|
| <code>log</code> | Adds a line to the system log. |

**15.2.17. Maximus Control File Information**

The functions listed in Table 15.29 read from the Maximus .PRM file.

**Table 15.29 Control File Functions**

| Function                | Description                             |
|-------------------------|---|
| <code>prm_string</code> | Returns a string from the MAX.PRM file. |

**15.2.18. Privilege Level Information**

The functions listed in Table 15.30 return information about user classes (from the class definition file).

**Table 15.30 Privilege Level Functions**

| Function                     | Description  |
|------------------------------|--|
| <code>class_abbrev</code>    | Returns the abbreviation for a user class.               |
| <code>class_info</code>      | Returns information about a user class.                  |
| <code>class_loginfile</code> | Returns the login filename for a user class.             |
| <code>class_name</code>      | Returns the name for a user class.                       |
| <code>class_to_priv</code>   | Converts a user class to a privilege level.              |
| <code>privok</code>          | Checks if the user's access level satisfies a given ACS. |

**15.2.19. Chat and Multinode Chat**

The functions listed in Table 15.31 provide a very limited interface to the system and multinode chat facilities.

**Table 15.31 Chat Functions**

| Function                      | Description                                |
|-------------------------------|--|
| <code>chat_querystatus</code> | Query the multinode chat status of a user. |
| <code>chatstart</code>        | Enable local chat mode.                    |

**15.2.20. Multilingual Support**

The functions listed in Table 15.32 allow strings to be retrieved from the language file.

**Table 15.32 Multilingual Functions**

| Function | Description                                     |
|----------|---|
| hstr     | Extract a string from a user language heap.     |
| lstr     | Extract a string from the system language file. |

**15.2.21. Static data**

The functions listed in Table 15.33 allow data to be created that stays “alive” for an entire Maximus session.

**Table 15.33 Static Data Functions**

| Function              | Description                                 |
|-----------------------|---|
| create_static_data    | Creates a static data object.               |
| create_static_string  | Creates a static string.                    |
| destroy_static_data   | Destroys a static data object.              |
| destroy_static_string | Destroys a static string.                   |
| get_static_data       | Gets information from a static data object. |
| get_static_string     | Gets information from a static string.      |
| set_static_data       | Stores information in a static data object. |
| set_static_string     | Stores information in a static string.      |

**15.2.22. ANSI and RIPscrip Support**

The functions listed in Table 15.34 are used to query ANSI and RIPscrip support and to perform RIPscrip-specific file functions. (For information on sending ANSI or AVATAR cursor control codes to the user, instead see section 13.5.)

**Table 15.34 ANSI and RIPscrip Functions**

| Function    | Description  |
|-------------|--|
| ansi_detect | Determines if the user’s terminal supports ANSI.                           |
| rip_detect  | Determines if the user’s terminal supports RIPscrip.                       |
| rip_hasfile | Determines if the user already has a specific RIPscrip scene or icon file. |
| rip_send    | Sends a RIPscrip file to the user.   |

## 15.3. Function Descriptions

This section contains detailed descriptions of all of the functions contained in the MEX run-time library.

As a reminder, do not forget to include the following line at the beginning of any MEX source file that accesses the library routines given in this section:

```
#include <max.mh>
```

---

### ansi\_detect

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int ansi_detect();   |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | TRUE if the user's terminal supports ANSI; FALSE otherwise.  |
| <b>Description</b>  | <p>The <b>ansi_detect</b> function is used to query the remote terminal to determine if it supports ANSI graphics. If the remote terminal reports that ANSI graphics are supported, this function returns TRUE.</p> <p>Note that some terminal programs may not support the control sequence used to query ANSI support. This means that this function may sometimes return FALSE, even if the user's terminal program does support ANSI. However, a return value of TRUE always indicates that the user's terminal supports ANSI.</p> <p>After the user has logged on, the user's ANSI graphics preference can be read from the <i>usr.video</i> field.</p> |

---

### call\_close

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | void call_close();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | None  |
| <b>Description</b>  | <p>This function releases the resources that were allocated to a MEX program by the <b>call_open</b> function. <b>call_close</b> should always be called when the application is finished using the caller log.</p> |

---

**call\_numrecs**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | long call_numrecs();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | If the file was successful accessed, the number of records contained in the caller log; 0 otherwise.  |
| <b>Description</b>  | The <b>call_numrecs</b> function enumerates the records in the caller log file. The <b>call_read</b> function can be used to read any of the records in the caller file, up to and including the record number returned by this function. |

---

**call\_open**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int call_open();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | TRUE if the caller file was opened successfully; FALSE otherwise.   |
| <b>Description</b>  | <p><b>call_open</b> is used to access the caller log, as defined by the <b>File Callers</b> keyword in the system control file.</p> <p>The caller log is different from the system log file, although both files contain similar information. While the system log file contains an ASCII listing of system events, meant to be read by humans, the caller log contains binary information that can be read by MEX programs and third-party utilities.</p> <p>The <b>call_open</b> function is used to obtain access to the caller log. After <b>call_open</b> has been called, any of the other <b>call_*</b> functions can be used to retrieve information from the caller log.</p> |

---

**call\_read**


---

|                  |  |
|------------------|--|
| <b>Prototype</b> | int call_read(long: recno, <b>ref</b> struct _callinfo: ci);   |
| <b>Arguments</b> | <p><i>recno</i>      The record number to be read from the caller log file. A value of 0 specifies the first record in the file. The range of valid record numbers can be determined by calling <b>call_numrecs</b>. In general, the range is from 0 to <i>call_numrecs()-1</i> (inclusive).</p> |

*ci* The *\_callinfo* structure into which the found record is to be placed.

**Return Value** TRUE if the record was successfully read; FALSE otherwise

**Description** This function reads a record from the caller log file. The record specified by the *recno* parameter will be read from the caller log and placed into the structure referenced by the *ci* parameter.

The *\_callinfo* structure is defined in **max.mh**. A description of the fields in the structure can be found in the previous section.

**Example** A typical sequence of calls for accessing the caller file (without error-checking) is:

```
#include <max.mh>

int main()
{
    struct _callinfo: ci;
    long: num_recs;
    int: i;

    call_open();
    num_recs := call_numrecs();

    for (i := 0; i < num_recs; i := i+1)
    {
        call_read(i, ci);
        // Process information in the "ci" structure
    }

    return 0;
}
```

---

---

## carrier

---

---

**Prototype** int carrier();

**Arguments** None

**Return Value** TRUE if DCD is present (high); FALSE if DCD is not present (low). This function always returns TRUE for local sessions.

**Description** This function is used to sample the status of the modem DCD line. DCD is present if a caller is currently on-line.

In most cases, DCD will always be present, since Maximus will normally recycle if a caller hangs up. However, if the automatic carrier-checking feature has been dis-



abled with **dcd\_check**, the **carrier** function can be used to manually check the status of the DCD line.

---

## chat\_querystatus

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int chat_querystatus(ref struct _cstat: cs);   |
| <b>Arguments</b>    | <i>cs</i> A reference to a caller status structure. On input, the <i>cs.task_num</i> field contains the task number whose status is to be queried. On output, the rest of the fields in the structure will be updated to describe the status of the requested task.  |
| <b>Return Value</b> | TRUE if the status information was successfully read; FALSE if the task was not on-line, an invalid task number was specified, or if Maximus could not communicate with the chat server.   |
| <b>Description</b>  | <p>This function is used to retrieve the status of an on-line user, including the user's name, status, and chat availability flag.</p> <p>Before calling <b>chat_querystatus</b>, the <i>cs.task_num</i> field should be filled out with the task number of the task to be queried.</p> <p>After the call, the <i>cs</i> structure will be filled out with all of the information about the requested task number.</p> |

---

## chatstart

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | void chatstart();  |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | None   |
| <b>Description</b>  | <p>The <b>chatstart</b> function is used to inform Maximus that the user is entering "chat" mode. This function is typically only used for MEX-based chat programs.</p> <p>When chat mode is in effect, the user's time limit is not decremented, and the "chat requested" flag on the status line is reset.</p> |

---

**class\_abbrev**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string class_abbrev(int: priv);   |
| <b>Arguments</b>    | <i>priv</i> The privilege level to be queried. This number can be in the range 0 through 65535.   |
| <b>Return Value</b> | A string indicating the name of the user class associated with the specified privilege level.   |
| <b>Description</b>  | This function is used to display a textual representation of a privilege level, rather than the numeric value of the privilege level itself. This function uses the information provided in the access control file to translate numeric privilege levels into strings. |
| <b>Example</b>      | If the standard privilege levels have not been modified, the following code will display "SysOp." (100 is the privilege level of the SysOp class, as defined in the standard access control file.)  |

```
#include <max.mh>

int main()
{
    string: s;

    s := class_abbrev(100);
    print(s);

    return 0;
}
```

---

**class\_info**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | long class_info(int: priv, int: cit);  |
| <b>Arguments</b>    | <i>priv</i> Privilege level which is to be queried.<br><br><i>cit</i> One of the <i>CIT_*</i> constants describing the type of query to perform.                           |
| <b>Return Value</b> | This function returns the requested value from the class information structure, or -1 if an invalid value was specified for <i>cit</i> .                                   |
| <b>Description</b>  | This function is used to query a given privilege level for various types of information, such as its default time limits, download limits, and other miscellaneous values. |

Maximus first searches the definitions in the access control file to find the privilege class which is closest to (but which does not exceed) the privilege level specified by *priv*. It then returns the information requested by the *cit* parameter.

The *cit* parameter can be any of the constants described in Table 15.35 below:

**Table 15.35 Class Information Types**

| cit               | Description   |
|-------------------|---|
| CIT_NUMCLASSES    | If this value is specified, the <i>priv</i> parameter is ignored, and <i>class_info</i> returns the number of privilege level classes that are defined in the access control file.  |
| CIT_DAY_TIME      | The maximum daily time limit.   |
| CIT_CALL_TIME     | The maximum per-connection time limit.  |
| CIT_DL_LIMIT      | The daily download limit, in kilobytes.   |
| CIT_RATIO         | The maximum download : upload ratio.  |
| CIT_MIN_BAUD      | The minimum speed required to log on.   |
| CIT_MIN_XFER_BAUD | The minimum speed required to download or upload files.   |
| CIT_MAX_CALLS     | The maximum number of times that a user can call in a given day.  |
| CIT_FREE_RATIO    | The number of kilobytes that can be downloaded before the download ratio takes effect.  |
| CIT_UPLOAD_REWARD | Percentage of the user's time rewarded for uploading files.   |
| CIT_ACCESSFLAGS   | User access flags, as defined in the access control file. See the <i>CFLAGA_*</i> definitions in <i>max.mh</i> for more information.  |
| CIT_MAILFLAGS     | User mail capability flags, as defined in the access control file. See the <i>CFLAGM_*</i> definitions in <i>max.mh</i> for more information.   |
| CIT_USERFLAGS     | User-specific flags, which were specifically designed for use by MEX programs. See the documentation for these flags in the access control file for more information.   |
| CIT_LEVEL         | The numeric privilege level associated with this class. Since Maximus retrieves the class with the privilege level closest to (but not exceeding) the <i>priv</i> parameter, the value returned by this function will not necessarily be the same as the value provided for <i>priv</i> . |
| CIT_CLASSKEY      | The "key" associated with the class. This is normally the first letter of the class abbreviation, but it can be changed manually by the SysOp.  |
| CIT_INDEX         | The "index number" of the specified privilege level.  |

|             |  |
|-------------|--|
| CIT_OLDPRIV | The old-style Maximus 2.x privilege level that corresponds to priv. This can be used when writing programs to manipulate data structures that are used by external programs designed to work with Maximus 2.x. |
| CIT_BYINDEX | If this value is combined with any of the CIT_ parameters above (using the bitwise or operator), the value of priv is assumed to be a class index rather than a privilege level.                               |

---

This code displays the maximum daily time limit for the current user:

```
#include <max.mh>

int main()
{
    long: lim;

    lim := class_info(usr.priv, CIT_DAY_TIME);
    print("Maximum daily limit: ", lim, " minutes.\n");

    return 0;
}
```

The following code can be used to iterate through all of the class definitions in the access control file:

```
#include <max.mh>

int main()
{
    int: i, n;
    unsigned int: priv;

    n := class_info(0, CIT_NUMCLASSES);

    for (i := 0; i < n; i := i+1)
    {
        priv := class_info(i,
                           CIT_LEVEL | CIT_BYINDEX);
        print("Priv for class ", i, " is ",
              priv, '\n');
    }

    return 0;
}
```

---

**class\_loginfile**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | string class_loginfile(int: priv);   |
| <b>Arguments</b>    | <i>priv</i> The privilege level to be queried  |
| <b>Return Value</b> | A string representing the log-in file specific to callers in this privilege level. The null string ("") is returned if no log-in file is defined for the requested privilege level.  |
| <b>Description</b>  | <p>This function is used to retrieve the name of the custom log-in file for members of a specific privilege level. The custom log-in file is typically used to display information that only pertains to a specific group of users.</p> <p>While Maximus will normally display this file automatically at log-on, MEX programs can also use this information to display the file manually.</p> |

**Example**      This code will display the custom log-on file for the current user:

```
#include <max.mh>

int main()
{
    string: file;
    char: nonstop;

    file := class_loginfile(usr.priv);
    nonstop := 0;

    if (file <> "")
        display_file(file, nonstop);

    return 0;
}
```

---

**class\_name**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | string class_name(int: priv);  |
| <b>Arguments</b>    | <i>priv</i> The privilege level to be queried  |
| <b>Return Value</b> | A string containing the name of the privilege level.   |
| <b>Description</b>  | This function is used to obtain the description for a specific privilege level from the access control file. |

---

---

## class\_to\_priv

---

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | unsigned int class_to_priv(string: classabbrev);  |
| <b>Arguments</b>    | <i>classabbrev</i> A string containing an abbreviation for a class privilege level.   |
| <b>Return Value</b> | The numeric privilege level associated with the name, or 65535 if the name could not be matched to any existing class.  |
| <b>Description</b>  | This function is used to translate text-based privilege level names into numeric privilege level values, such as those stored in many of the internal Maximus structures. |

---

---

## close

---

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int close(int fd)  |
| <b>Arguments</b>    | <i>fd</i> A file handle previously returned by <b>open</b> .   |
| <b>Return Value</b> | TRUE if the file was successfully closed; FALSE otherwise.   |
| <b>Description</b>  | This function is used to close a file handle that was previously opened by <b>open</b> . MEX programs should close file handles as soon as all file operations on that handle have been completed. |

---

---

## compressor\_num\_to\_name

---

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string compressor_num_to_name(int: compressor);   |
| <b>Arguments</b>    | <i>compressor</i> An integer index for a compression program in <b>compress.cfg</b> .                                     |
| <b>Return Value</b> | A string containing the name of the compressor, or the null string ("") if the compressor index is invalid.               |
| <b>Description</b>  | This function is used to translate the <i>usr.compress</i> field in the user structure to obtain a human-readable string. |

---

## create\_static\_data

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int create_static_data(string: key, long: size);</code>   |
| <b>Arguments</b>    | <p><i>key</i>        A string containing a used-defined key. This key identifies the data item to be created. This string must be used to identify this data item in all future calls to the <b>*_static_data</b> functions. The recommended format for the <i>key</i> string is:</p> <p style="text-align: center;"><i>“program_name : program_specific_value”</i>, where <i>program_name</i> is the name of the MEX program being executed, and <i>program_specific_value</i> is a brief description of the item to be stored.</p> <p><i>size</i>        The size of the data object to be created. The value used for this parameter is normally obtained by using the <b>sizeof</b> operator on the type of the object to be stored.</p>  |
| <b>Return Value</b> | <p>0 if the data object was created successfully;<br/>         -1 if not enough memory was available to satisfy the request;<br/>         -2 if an invalid parameter was specified;<br/>         -3 if the key name already exists.</p>   |
| <b>Description</b>  | <p>Normal MEX variables are destroyed as soon as the calling MEX program returns to Maximus, but the <b>create_static_data</b> function allows programs to create persistent data objects that remain around for an entire Maximus session.</p> <p>This function is used to store integral data types (<b>char</b>, <b>int</b> and <b>long</b>) and aggregates (user-defined <b>struct</b> and <b>array</b> types). To manipulate strings, see <b>create_static_string</b>.</p> <p>The <i>size</i> parameter should indicate the size of the data to be stored within the static data object. The <b>sizeof</b> operator is normally used to calculate this value. For example, if a <b>long</b> is to be stored, <i>size</i> should be set to <b>sizeof(long)</b>.</p> <p>Static data objects created by <b>create_static_data</b> are initially set to binary zeroes.</p> |
| <b>Example</b>      | <p>The following code demonstrates how to create a static data object called “myfoo” from <i>test.mex</i>. The data object is large enough to hold a “foo” structure:</p> <pre> struct foo {     int:  bar;     int:  boz; };  // ...  struct foo: myfoo; </pre>  |

```
create_static_data("test:current_foo",
                  sizeof(struct foo));
```

---

## create\_static\_string

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int create_static_string(string: key);</code>  |
| <b>Arguments</b>    | <p><i>key</i> A string containing a used-defined key. This key is used to identify the data item to be created. This string must be used to identify this data item to all future calls to the <b>*_static_data</b> functions. The recommended format for the <i>key</i> string is:</p> <p style="text-align: center;"><i>“program_name : program_specific_value”</i>, where <i>program_name</i> is the name of the MEX program being executed, and <i>program_specific_value</i> is a brief description of the string to be stored.</p> |
| <b>Return Value</b> | <p>0 if the string was created successfully;<br/>         -1 if not enough memory was available to satisfy the request;<br/>         -2 if an invalid parameter was specified;<br/>         -3 if the key name already exists.</p>   |
| <b>Description</b>  | <p>This function is similar to <b>create_static_data</b>, except that it creates persistent string variables. Maximus will manage the string size internally, so no <i>size</i> parameter is required for <b>create_static_string</b>.</p> <p>A string created by <code>create_static_string</code> is initially empty.</p> <p>For more information, see the description for <b>create_static_data</b>.</p>  |
| <b>Example</b>      | <p>This code shows how to create a static string called “mystring” from <b>test.mex</b>:</p>   |

```
create_static_string("test:mystring");
```

---

## dcd\_check

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int dcd_check(int: state);</code>  |
| <b>Arguments</b>    | <p><i>state</i> A flag indicating whether or not DCD checking is to be used. If non-zero, DCD checking is performed. If zero, DCD checking is not performed.</p>   |
| <b>Return Value</b> | <p>An integer describing the prior state of DCD checking. A value of 1 indicates that DCD checking was previously being performed; a value of 0 indicates that DCD checking was not being performed.</p> |



**Description** This function is used to control whether or not Maximus checks the DCD line. Normally, the absence of DCD indicates that the caller has hung up. However, some special MEX programs may wish to ignore this signal, such as in call-back verifier programs.

When automatic DCD checking is off, the **carrier** function can be used to manually check for DCD.

---

## destroy\_static\_data

---

**Prototype** `int destroy_static_data(string: key);`

**Arguments** *key* The key for the static data object to be destroyed. This must be the same as the *key* parameter passed to **create\_static\_data**.

**Return Value** 0 if the data object was destroyed successfully;  
-1 if the key name was not found

**Description** This function destroys a static object key and the corresponding object data, and it then returns the associated memory to Maximus. This code must be called whenever a static data object is no longer required.

**Example** To destroy the static data object that was created in the example for the **create\_static\_data** function:

```
destroy_static_data("test:current_foo");
```

---

## destroy\_static\_string

---

**Prototype** `int destroy_static_string(string: key);`

**Arguments** *key* The key for the static string to be destroyed. This must be the same as the *key* parameter passed to **create\_static\_string**.

**Return Value** 0 if the string was destroyed successfully;  
-1 if the key name was not found

**Description** This function destroys a static string key and the corresponding string data, and it then returns the associated memory to Maximus. This code must be called whenever a static string is no longer required.

**Example** To destroy the static string that was created in the example for the **create\_static\_string** function:

```
destroy_static_string("test:mystring");
```

---

## display\_file

---

**Prototype**      `int display_file(string: filename, ref char: nonstop);`

**Arguments**      *filename*    The name of the **.bbs** file to be displayed to the user. If no extension is provided, “.bbs” is assumed.

### WARNING!

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain `“\\max\\misc\\foo.bbs”`.

*nonstop*    A reference to a variable used for controlling non-stop display action. See the description and examples below for more information.

**Return Value**    0 if the file was displayed successfully;  
-1 otherwise.

**Description**    This function is used to display a **.bbs** file from within a MEX program. Any **.bbs** file can be displayed, as long as it does not try to recursively call another MEX program.

The *nonstop* parameter is used to control non-stop file display. In most cases, this variable should be initialized to 0 before calling **display\_file** and then left alone.

If *nonstop* is set to 0 before calling **display\_file**, normal “More” processing will take effect. Maximus will prompt the user with “More [Y,n,=]” after every page of information has been displayed. If the user selects “=” at a More prompt, the *nonstop* parameter will be updated with a value of 1.

If *nonstop* is set to 1 before calling **display\_file**, Maximus will assume that the user already asked for non-stop display, so it will not prompt the user after every page.

The *nonstop* variable can be used to implement the non-stop display of multiple files at a time.

**Example**          This code shows how to display a single file:

```
#include <max.mh>

int main()
{
    char: nonstop;
```

```

        nonstop := 0;
        display_file("c:\\max\\misc\\logo.bbs", nonstop);

    return 0;
}

```

This code shows how to display multiple files in succession, allowing for continuous display:

```

#include <max.mh>

int main()
{
    char: nonstop;

    nonstop := 0;        // only initialize to 0 for
                        // the first display

    display_file("c:\\max\\misc\\logo.bbs", nonstop);
    display_file("c:\\max\\misc\\welcome.bbs",
                nonstop);

    return 0;
}

```

The above code would display **c:\max\misc\logo.bbs** to the user, followed by **c:\max\misc\welcome.bbs**. If the user requested non-stop output while displaying **logo.bbs**, the **welcome.bbs** file would also be displayed in non-stop mode. If this is not desired, *nonstop* should be set to 0 before each call to **display\_file**.

---

## do\_more

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int do_more( <b>ref</b> char: nonstop, string: color);   |
| <b>Arguments</b>    | <p><i>nonstop</i>    A reference to a variable used to store the non-stop display status. This variable is normally initialized by a call to the <b>reset_more</b> function.</p> <p><i>color</i>       A string containing the color to be used for displaying the More prompt. (This is normally one of the <i>COL_*</i> constants from <b>mex.mh</b>.)</p> |
| <b>Return Value</b> | TRUE if less than a page of information has been displayed since the last <b>reset_more</b> call, if the user answered “yes” or “=” at the prompt displayed by <b>do_more</b> , or if non-stop display is in effect; FALSE otherwise.  |

**Description**

This function is used to display “More [Y,n,=]” prompts to the user in appropriate places. This is useful when displaying a long list of information that is longer than one screen.

The **do\_more** function keeps track of the number of lines that have been displayed since the last **reset\_more** call. If the line count is less than the length of the screen, **do\_more** will do nothing and return TRUE.

If the line count is greater than or equal to the length of the screen, **do\_more** will display a “More [Y,n,=]” prompt and reset the displayed line count, as long as the *nonstop* variable has a value of 0:

If the user answers “N” at the prompt, **do\_more** will return FALSE.

If the user answers “Y” at the prompt, **do\_more** will return TRUE.

If the user answers “=” at the prompt, **do\_more** will return TRUE and set the value of *nonstop* to 1.

However, **do\_more** will not display a More prompt to the user if the *nonstop* variable contains a value of 1 upon entry to the **do\_more** function.

For a final special case, if the *nonstop* variable has a value of -1, **do\_more** will always prompt the user for more, as in the case where *nonstop* was 0, as above. However, the value of *nonstop* will never be changed. Consequently, setting *nonstop* to -1 ensures that the More prompt will be displayed to the user after every page, even if the user specifically selects the “=” option.

**Example**

For example, to display output with paging support provided by the **do\_more** function:

```
#include <max.mh>

int main()
{
    int: line;
    int: done;
    char: nonstop;

    reset_more(nonstop);
    done := 0;

    for (line := 1;
        line <= 100 and done=0;
        line := line + 1)
    {
        print("Line #", line, '\n');
```

```
        if (do_more(nonstop, COL_WHITE) = 0)
            done := TRUE;
    }
    return 0;
}
```

---

---

### file\_area

---

---

|              |   |
|--------------|---|
| Prototype    | void file_area();   |
| Arguments    | None  |
| Return Value | None  |
| Description  | This function displays the file area menu. Calling <b>file_area</b> is equivalent to calling <i>menu_cmd(MNU_FILE_AREA, "")</i> , although calling <b>file_area</b> is slightly faster. |

---

---

### fileareafindclose

---

---

|              |   |
|--------------|---|
| Prototype    | void fileareafindclose();   |
| Arguments    | None  |
| Return Value | None  |
| Description  | The <i>fileareafindclose</i> function terminates an existing <b>fileareafindfirst</b> search. This function should be called whenever the program has finished using the file area data file. |

---

---

### fileareafindfirst

---

---

|           |  |
|-----------|--|
| Prototype | int fileareafindfirst( <b>ref</b> struct _farea: fa, string: name, int: flags);  |
| Arguments | <p><i>fa</i>            A reference to a file area information structure. If this function finds a file area matching the <i>name</i> and <i>flags</i> parameters, information about that file area will be placed in this structure.</p> <p><i>name</i>        Name of a specific file area to find. If <i>name</i> is the null string (""), this function will find the first available file area.</p> |

*flags* A flag indicating whether or not **FileDivisionBegin** and **FileDivisionEnd** records are to be returned. If this flag is equal to *AFFO\_DIV*, division records will be returned. Otherwise, if this flag is equal to *AFFO\_NODIV*, division records will be skipped.

**Return Value** TRUE if an area was found; FALSE otherwise.

**Description** The **fileareafindfirst** function searches for a specific file area, as specified by the *name* parameter. It also finds division records within the area file if the *AFFO\_DIV* flag is specified.

Only areas which can be accessed by the user will be returned by this function. This function will also skip file areas that have the **Type Hidden** style.

**Example** The following code will display a list of all file areas:

```
#include <max.mh>

int main()
{
    struct _farea: fa;

    if (fileareafindfirst(fa, "", AFFO_NODIV))
    {
        do
        {
            print("Area: ", fa.name, '\n');
        }
        while (fileareafindnext(fa));

        fileareafindclose();
    }

    return 0;
}
```

---

---

## fileareafindnext

---

---

**Prototype** int fileareafindnext(**ref** struct \_farea: fa);

**Arguments** *fa* A reference to the file area structure to be updated with file area information. This structure must have been originally filled in by a **fileareafindfirst**, **fileareafindnext**, or **fileareafindprev** call.

**Return Value** TRUE if another file area was found; FALSE if no file area could be found.

**Description** The **fileareafindnext** function finds the next file area that is accessible to the user. The search is carried out from the position where the last **fileareafindnext**, **fileareafindprev**, or **fileareafindfirst** call terminated. Consequently, if the last **fileareafind\*** call returned information for area “X”, this function would return information for the next user-accessible area following area “X”.

---

## fileareafindprev

---

**Prototype** int fileareafindprev(ref struct \_farea: fa);

**Arguments** *fa* A reference to the file area structure to be updated with file area information. This structure must have been originally filled in by a **fileareafindfirst**, **fileareafindnext**, or **fileareafindprev** call.

**Return Value** TRUE if another file area was found; FALSE if no file area could be found.

**Description** The **fileareafindprev** function finds the previous file area that is accessible to the user. The search is carried out from the position where the last **fileareafindnext**, **fileareafindprev**, or **fileareafindfirst** call terminated. Consequently, if the last **fileareafind\*** call returned information for area “X”, this function would return information for the first user-accessible area that precedes area “X”.

---

## fileareaselect

---

**Prototype** int fileareaselect(string: name);

**Arguments** *name* Name of the file area to be selected as the user’s current file area.

**Return Value** TRUE if the area was successfully selected; FALSE otherwise.

**Description** The **fileareaselect** function is used to change the user’s current file area. Upon return, this function also updates the global *farea* variable with information about the new file area.

---

## filecopy

---

**Prototype** int filecopy(string: old, string: new);

**Arguments** *old* Name of the file to be copied.

**WARNING!**

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain “`\\max\\misc\\foo.bbs`”.

*new* Target path and filename for the copy.

**Return Value** TRUE if the file was successfully copied; FALSE otherwise.

**Description** This function copies a file from *old* to *new*.

---

**filedate**

---

**Prototype** `int filedate(string: filename, ref struct stamp: fdate);`

**Arguments** *filename* The name of the file to be queried.

**WARNING!**

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain “`\\max\\misc\\foo.bbs`”.

*fdate* A reference to a *stamp* structure that will be updated to contain information about the file’s date.

**Return Value** This function returns TRUE if the file date was successfully obtained; FALSE otherwise.

**Description** The **filedate** function is used to obtain the last-write date for a specific file.

---

**fileexists**

---

**Prototype** `int fileexists(string: filename);`

**Arguments** *filename* The name of the file to be queried.

**WARNING!**



All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain `"\\max\\misc\\foo.bbs"`.

**Return Value** TRUE if the file exists; FALSE otherwise.

**Description** This function is used to determine whether or not the specified file exists on disk.

---

## filefindclose

---

**Prototype** void filefindclose(**ref** struct \_ffind: ff);

**Arguments** *ff* A reference to the *\_ffind* structure that was returned by a previous call to *filefindfirst*.

**Return Value** None

**Description** This function releases the system resources that were allocated when performing a **filefindfirst** call.

---

## filefindfirst

---

**Prototype** int filefindfirst(**ref** struct \_ffind: ff, string: filename, int: attribs);

**Arguments** *ff* A reference to a *\_ffind* structure to be updated by this function. Upon return, this structure is updated with information about the file found (if any).

*filename* A wildcard specification (or individual filename) for which this function is to search.

*attribs* A set of attributes used to describe the types of files to find. This should be *FA\_NORMAL* in most cases, but it can also be one or more of the following, connected using the *bitwise or* operator: *FA\_READONLY*, *FA\_HIDDEN*, *FA\_SYSTEM*, *FA\_VOLUME*, *FA\_SUBDIR*, or *FA\_ARCHIVE*. This mask restricts the filenames returned to those which have the specified attributes.

**Return Value** TRUE if a file was successfully found; FALSE otherwise.

**Description** This function is normally used to expand file or directory wildcards. Given a filename specification such as `"A*.*"`, the **filefindfirst** function will search for the specified file using the host operating system.

The first file matching *filename* will be returned in the *ff* structure, including information about the file's name, size, date and attributes. (However, for finding file information for a single, non-wildcard filename, the **filesize** and **fileexists** functions are generally much faster.)

To find the second and subsequent files that match the specified wildcard, see the **filefindnext** function.

### Example

This code displays all of the files in the current directory:

```
#include <max.mh>

int main()
{
    struct _ffind: ff;

    if (filefindfirst(ff, " *.*", FA_NORMAL))
    {
        do
        {
            print("Found file: ",
                ff.filename, '\n');
        }
        while (filefindnext(ff));

        filefindclose(ff);
    }

    return 0;
}
```

---

---

## filefindnext

---

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int filefindnext( <b>ref</b> struct _ffind: ff);   |
| <b>Arguments</b>    | <i>ff</i> A reference to the <i>_ffind</i> structure that was returned by a previous call to <i>filefindfirst</i> .  |
| <b>Return Value</b> | TRUE if another file was found; FALSE otherwise.   |
| <b>Description</b>  | The <b>filefindnext</b> function continues a search that was started by <b>filefindfirst</b> . This function returns the next filename matching the pattern specified in the original <b>filefindfirst</b> call. |

---

**filesize**


---

**Prototype**      `long filesize(string: filename);`

**Arguments**      *filename*    The filename whose size is to be queried.

**WARNING!**

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain “`\\max\\misc\\foo.bbs`”.

**Return Value**      -1 if the file does not exist; otherwise, this function returns the file size.

**Description**      The **filesize** function is used to retrieve the size of a file.

---

**get\_static\_data**


---

**Prototype**      `int get_static_data(string: key, ref void: data);`

**Arguments**      *key*            The key for the static data object to be retrieved. This must be the same as the *key* parameter originally passed to **create\_static\_data**.

*data*            A reference to the local data object that is to be updated with a copy of the static data object. Note that this parameter is a *void reference*, meaning that any type of object (char, int, long, array or struct) can be referenced.

**Return Value**      0 if the data object was retrieved successfully;  
                      -1 if the key name was not found.

**Description**      The **get\_static\_data** function is used to retrieve data that was previously stored by the **set\_static\_data** function. If *key* is a valid static data key that was created by **create\_static\_data**, this function will retrieve the information in the static data object and copy it into the local variable referenced by *data*.

**Example**            The following code creates a static data object, writes a value to it, and then retrieves it again (without error checking):

```
#include <max.mh>

int main()
{
    long: l1, l2;
    string: dataname;
```

```

    dataname := "test:mylong";
    create_static_data(dataname, sizeof(long));
    l1 := 1234;
    set_static_data(dataname, l1);

    // Do some other things here, or even exit back
    // to Maximus and restart the MEX program.

    get_static_data(dataname, l2);
    print("l2 = ", l2, '\n');
    return 0;
}

```

---

## get\_static\_string

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int get_static_string(string: key, ref string: data);</code>  |
| <b>Arguments</b>    | <p><i>key</i>        The key for the static string to be retrieved. This must be the same as the <i>key</i> parameter passed to <b>create_static_string</b>.</p> <p><i>data</i>       A reference to the string to be updated with a copy of the static string.</p>   |
| <b>Return Value</b> | 0 if the string was retrieved successfully;<br>-1 if the key name was not found.  |
| <b>Description</b>  | The <b>get_static_string</b> function is used to retrieve strings that were previously stored by the <b>set_static_string</b> function. If <i>key</i> is a valid static string key that was created by <b>create_static_string</b> , this function will retrieve the information in the static string and copy it into the local string referenced by <i>data</i> . |
| <b>Example</b>      | The following code creates a static string, writes a value to it, and then retrieves it again (without error checking):   |

```

#include <max.mh>

int main()
{
    string: s1, s2;
    string: stringname;

    stringname := "test:mystring";
    create_static_string(stringname);
    s1 := "foo";
    set_static_string(stringname, s1);

    // Do some other things here, or even exit back
    // to Maximus and restart.
}

```

```

    get_static_string(stringname, s2);
    print("s2 = '", s2, "'\n");
    return 0;
}

```

---

## getch

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | char getch();   |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | The character entered by the user.  |
| <b>Description</b>  | <p>This function performs raw character input. It simply returns the character typed by the user. If no character is available, Maximus will wait until the user presses a key.</p> <p>All of the normal Maximus input routines are used, so if the system operator enables local keyboard input, characters entered at the local console will also be retrieved by this function.</p> <p>Note that this function does not perform any input processing. Regardless of the user's hotkey setting, IBM characters setting, or other flags, Maximus will simply return the character received over the modem. This includes scan codes and raw key codes from terminal programs running in "doorway mode" (and from the local console).</p> <p>Except in special cases where raw keyboard input is required, the <b>input_ch</b> function should be used instead of this function. <b>input_ch</b> provides access to formatted data input, including command stacking for non-hotkey users, prompts, and more.</p> |

---

## hstr

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string hstr(ref string: heapname, int: index);  |
| <b>Arguments</b>    | <p><i>heapname</i> Name of the heap from which the string is to be fetched.</p> <p><i>index</i> Index number (within the heap) of the string to be fetched.</p> |
| <b>Return Value</b> | The string that matches the requested heap name and index, or the null string ("") if the heap/index pair could not be found.                                   |

**Description** The *hstr* function is used to retrieve strings from MEX-specific language files. The *heapname* and *index* parameters combine to form a key that refers to a unique string in one of the user-defined language heaps in **english.mad**.

This function is only used to retrieve MEX-specific strings from **english.mad**. These strings are always contained in heaps that start with an “=” character, rather than the “:” character that is used to begin the standard system language heaps.

Calls to **hstr** are normally generated automatically by MAID when it creates the **english.mh** include file for MEX programs. When MAID parses a language file, it gathers information about all of the user-defined heaps. It then translates the name in front of each string into a MEX **#define** directive, which expands into a call to the **hstr** function using the correct *heapname* and *index* parameters.

---

## input\_ch

---

**Prototype** int input\_ch(int: type, string: options);

**Arguments** *type* This parameter specifies a number of options that control the character input routine. Zero or more of the *CINPUT\_\** constants can be specified, combined using the *bitwise or* operator. For a detailed list of *CINPUT\_\** constants, see the description below.

*options* This string specifies an optional list of parameters. The contents of this string vary based on the settings used for *type*, as described below.

**Return Value** This function returns the character that is entered by the user.

**Description** The **input\_ch** function performs formatted character input. It calls the standard Maximus character input routines to get a key from the user. Among other things, this means that:

- Hotkey mode is properly handled. If the user has hotkeys enabled, the function will return as soon as a key is pressed. Otherwise, Maximus will accept a line of input and only return the first character.
- Standard Maximus input functionality can also be enabled, such as prompt display and <ctrl-c> handling.
- Command stacking is supported.

The *type* field controls how the input function operates. The field can be a combination of one or more of the values shown in Table 15.36. Multiple values can be combined using the *bitwise or* operator.

**Table 15.36 Character Input Types**

| Type              | Description   |
|-------------------|---|
| CINPUT_DISPLAY    | The character entered by the user should always be displayed, even if hotkey mode is enabled. This option is implied for the <code>input_list</code> function.  |
| CINPUT_ACCEPTABLE | Restrict the characters input by the user to the list of characters specified in the options string. For example, if the <code>CINPUT_ACCEPTABLE</code> flag is specified, and if options contains “abeq”, the user will only be able to enter an “A,” “B,” “E” or “Q” at the prompt.   |
| CINPUT_PROMPT     | Display the prompt contained in options before asking for input.  |
| CINPUT_SCAN       | Accept scan codes. This option instructs Maximus to return scan codes produced by function keys, cursor keys, and other special characters. (These scan codes can be created by pressing the appropriate function and cursor keys on the local SysOp keyboard, and they can also be created by remote users who use terminal programs in “DoorWay mode.”)<br>Although standard ASCII characters (such as letters, numbers, and punctuation) are still returned as a single character, when a function key or other non-ASCII character is pressed, <code>input_key</code> will return 0. In the following call to <code>input_key</code> , the scan code of the key will be returned. |
| CINPUT_NOXLT      | Do not translate special characters, such as carriage returns and newlines, into their ASCII equivalents. (For example, unless this option is used, <enter> will be returned as “.”) This option is implied for the <code>input_list</code> function.   |
| CINPUT_NOCTRLC    | Do not allow the user to press <ctrl-c> to abort the current entry and redisplay the prompt.  |
| CINPUT_P_CTRLC    | The prompt specified in options will not be initially displayed; the prompt is only displayed if the user presses <ctrl-c>.   |
| CINPUT_NOLF       | Do not display a linefeed after the user’s input character selection is displayed.  |
| CINPUT_FULLPROMPT | Do not add a bracketed list of acceptable characters to the prompt string. This option is only valid when used as parameter for the <code>input_list</code> function.   |
| CINPUT_ALLANSWERS | Allow the user to exit by pressing only <enter>, even if <code>CINPUT_ACCEPTABLE</code> is also specified.  |
| CINPUT_DUMP       | Flush the output buffer when a character is re-   |

|                             |   |
|-----------------------------|---|
|                             | ceived. This option is mostly useful in hotkey mode. This option is also implied for the <code>input_list</code> function.  |
| <code>CINPUT_NOUPPER</code> | Do not convert received characters to uppercase.  |
| <code>CINPUT_AUTOP</code>   | Display the prompt, even if the user has hotkeys enabled.   |
| <code>CINPUT_ANY</code>     | Any response to this function is valid, even if not contained in list. This option is only valid when used as a parameter for the <code>input_list</code> function. |

## input\_list

### Prototype

```
int input_list(string: list, int: type, string: help_file, string: invalid,
              string: prompt);
```

### Arguments

*list* This string contains a list of acceptable input characters. To allow a character that is not present in *list*, include the `CINPUT_ANY` option in the *type* parameter. The first uppercase character in the string will be chosen as the default option and will be selected if the user presses `<enter>` at the prompt.

*type* This parameter specifies a number of options that control the character input routine. Zero or more of the `CINPUT_*` constants can be combined using the *bitwise or* operator. For a detailed list of acceptable `CINPUT_` constants, see the description of the `input_ch` function, above.

*help\_file* If not a blank string, this specifies the name of the help file to be displayed when a question mark (“?”) is entered by the user. If this is not a blank string, Maximus will also add a “=help” to the end of *prompt*.

*invalid* This string is displayed to the user when an invalid character is entered.

*prompt* This string contains a prompt to be displayed to the user. Unless the `CINPUT_FULLPROMPT` option is included in the *type* parameter, a bracketed list of acceptable option letters is automatically added to the end of this string.

### Return Value

This function returns the option character entered by the user.

### Description

The `input_list` function prompts the user to enter a character from a predefined set of options. A typical prompt generated by `input_list` looks something like this:

```
Do you prefer lettuce, cabbage or broccoli [L,c,b]?
```

The first part of the prompt, “Do you prefer lettuce, cabbage or broccoli,” is provided in the *prompt* parameter.



The second part of the prompt, “[L,c,b]?” is automatically added by **input\_list**. By specifying the valid input characters in the *list* string as “Lcb,” the **input\_list** function will automatically format the “[L,c,b]” text and add the final question mark. Since the “L” is uppercase, it will be chosen as the default option if the user presses <enter>.

### Example

The following code implements the prompt described above:

```
#include <max.mh>

int main()
{
    int: ch;

    ch := input_list("Lcb",
                    0,
                    "",
                    "That is an invalid type of "
                    "produce. ",
                    "Do you prefer lettuce, "
                    "cabbage or broccoli");

    return 0;
}
```

---

## input\_str

---

### Prototype

int input\_str(**ref** string: s, int: type, char: ch, int: max, string: prompt);

### Arguments

*s*            On return, this variable will be updated to contain the text entered by the user.

*type*        This parameter contains one or more *INPUT\_\** options, combined using the *bitwise or* operator, which control how Maximus reads input from the user. These options are described in more detail in the function description, below.

*ch*           This special-purpose parameter is only used when a certain subset of the *INPUT\_\** options are specified in the *type* field, as indicated in the table below. Otherwise, this parameter should be 0.

*max*          The maximum length of the string to be returned in *s*.

*prompt*      The prompt to be displayed to the user before requesting input.

### Return Value

This function returns the length of the string placed in *s*.

**Description**

The **input\_str** function displays a prompt and waits for the user to enter a word or a string. This string is returned in the *s* parameter where it can be later examined by the MEX program.

The *type* parameter specifies a number of options that control how the input function reacts to user input. It also controls whether or not command accepts a word or a string, and whether or not command stacking is allowed.

The *type* parameter must contain exactly one of the following mutually-exclusive options: *INPUT\_LB\_LINE*, *INPUT\_NLB\_LINE*, and *INPUT\_WORD*. All of the other parameters in Table 15.37 are optional and can be specified in any combination.

**Table 15.37 Line Input Types**

| Type            | Description  |
|-----------------|--|
| INPUT_LB_LINE   | Read an entire line of input from the user. Command stacking is enabled, so if any unused input is in the stacking buffer (accessible as the global variable <code>input</code> ), it will be returned without displaying the prompt or asking the user for more input.  |
| INPUT_NLB_LINE  | Read an entire line of input from the user. Command stacking is disabled, so the contents of the line buffer (accessible as the global variable <code>input</code> ) are ignored. The prompt is always displayed and the user is always asked for input.   |
| INPUT_WORD      | Read a single word from the user. Command stacking is always enabled. If there is unused input in the stacking buffer (accessible as the global variable <code>input</code> ), the first word therein will be returned without displaying the prompt or asking the user for more input. In this context, a “word” is delimited by one or more spaces. If the stacking buffer is empty, the prompt will be displayed and Maximus will wait for the user to enter an entire string. The first word of this string (delimited by spaces) will be returned, and the rest of the string will remain in the line buffer. |
| INPUT_ECHO      | The character specified in <i>ch</i> is echoed back instead of the actual character typed by the user. This is useful for designing prompts for passwords or other sensitive information. This option and <i>INPUT_NOECHO</i> are mutually exclusive. (If neither option is specified, Maximus will echo the characters normally.)   |
| INPUT_NOECHO    | Do not echo any characters back to the remote. This option and <i>INPUT_ECHO</i> are mutually exclusive.   |
| INPUT_ALREADYCH | Pretend that the user has already entered the charac-  |

|                |  |
|----------------|--|
|                | ter given in <i>ch</i> . This is useful if an input sequence is chained off a hotkeyed menu option, or some other form of input that retrieves the first input character manually.   |
| INPUT_SCAN     | Allow scan codes to be placed in the returned string. See the description of CINPUT_SCAN in the input_ch function for more information on scan code formats.   |
| INPUT_NOCTRLC  | Do not allow the user to press <ctrl-c> to abort the current input and redisplay the prompt.   |
| INPUT_NOLF     | Do not send a linefeed after the user has finished entering the string.  |
| INPUT_WORDWRAP | Allow word-wrapping.   |
| INPUT_NOCLEOL  | Never issue clear-to-end-of-line (CLEOL) codes.  |
| INPUT_DEFAULT  | On input, pretend that the contents of <i>s</i> were already entered by the user. This option is useful if the first part of an input string is to be automatically generated. (However, the user can use the backspace key to modify this input.) On output, <i>s</i> will be updated with the full string by the user. |

---



---

## iskeyboard

---



---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int iskeyboard();   |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | TRUE if the local keyboard mode is active; FALSE otherwise.   |
| <b>Description</b>  | <p>This function is used to determine whether or not the local keyboard mode (toggled by the “A” character on the SysOp console) is active.</p> <p>Note that local sessions are always considered to be running in local keyboard mode.</p> |

---



---

## issnoop

---



---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int issnoop();                                 |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | TRUE if snoop mode is active; FALSE otherwise. |

**Description** This function determines whether or not Snoop mode is active. If snoop mode is enabled, the output sent to the remote user will also be echoed on the local screen. Snoop mode is enabled on the local console by pressing the “N” key.

---

## itostr

---

**Prototype** string itostr(int: i);

**Arguments** *i* The integer to be converted.

**Return Value** This function returns the string representation of the integer.

**Description** The **itostr** function is used to convert an integer to a string. The converted string contains the ASCII representation of the integer, from -32768 to 32767. See the **uitostr** function for converting unsigned integers.

This function is useful when strings must be mixed with the results of integral computations.

**Example** This code concatenates a string and a converted integer:

```
#include <max.mh>

int main()
{
    int: i1;
    string: s;

    i1 := 1000;
    s := "MEX is used by " + itostr(i1) + "s of "
        + "programmers";

    print(s);
    return 0;
}
```

---

## kbhit

---

**Prototype** int kbhit();

**Arguments** None

**Return Value** TRUE if a character is waiting; FALSE otherwise.

**Description** The **kbhit** function is used to determine if a character has been pressed by the remote user (or on the local console, if local keyboard mode is enabled).

If a character has been pressed, any of the **input\_str**, **input\_list**, **input\_ch** or **getch** functions can be used to retrieve the character.

---

## keyboard

---

**Prototype** `int keyboard(int: state);`

**Arguments** *state* The new state for keyboard mode. If this parameter is 1, local keyboard mode is enabled. If this parameter is 0, local keyboard mode is disabled.

**Return Value** The prior state of the local keyboard setting.

**Description** The **keyboard** function is used to set or reset the local keyboard mode. This function is primarily useful when a MEX program wishes to explicitly allow the SysOp to enter characters, even if local keyboard mode was originally off.

The return value of this function can be used at a later point in time to reset local keyboard mode to its original state.

---

## language\_num\_to\_name

---

**Prototype** `string language_num_to_name(int: lang);`

**Arguments** *lang* An integer index for a language defined in the language control file.

**Return Value** A string containing the name of the language, or the null string ("") if the language index is invalid.

**Description** This function is used to translate the *usr.lang* field in the user structure to obtain a human-readable string.

---

## localkey

---

**Prototype** `int localkey();`

**Arguments** None

**Return Value** TRUE if the last **getch** or **kbhit** returned a character that was entered on the SysOp console or by a local session; FALSE if the character was entered by a remote user.

**Description** The **localkey** function is used to determine the source of the most recent keystroke. This function can be useful if the processing of a character depends on whether the key was entered by a remote user or by the SysOp (or a caller logged on at the console).

---

---

## log

---

---

**Prototype** void log(string: text);

**Arguments** *text* Line to be placed in the system log. The first character of the string should indicate the priority of the log message, such as “!” or “#.” The second and following characters of the string represent the line to be logged.

**Return Value** None

**Description** This function adds the specified line to the Maximus system log file.

**Example** Given the following code:

```
log("!Could not find user record!");
```

The code above would create a log entry similar to the one shown below:

```
! 12 Jul 95 15:33:02 MAX Could not find user record!
```

---

---

## long\_to\_stamp

---

---

**Prototype** void long\_to\_stamp(long: time, **ref** struct \_stamp: st);

**Arguments** *time* A long integer containing the time value to be converted. The value contained in this parameter represents the number of seconds elapsed since January 1st, 1970 (UTC).

*st* A reference to a *\_stamp* structure. Upon return, this structure will be updated with values representing the time given by *time*.

**Return Value** None

**Description** This function is used to convert the result of the **time** function into a human-readable result. The values placed in the *st* structure correspond to the current date and time, expressed in terms of the current year, day, month, hour, minute and second.

---

---

## lstr

---

---

**Prototype** string lstr(int: index);

**Arguments** *index* An integer representing one of the strings in the *english.mad* language file.

**Return Value** The string specified by the *index* parameter, or the null string ("" ) if an invalid index number was specified.

**Description** The **lstr** function is used to retrieve a string from the standard strings in the system language file. (System language heaps are always declared in **english.mad** using the “:” character. To contrast, user-specific heaps — which are retrieved using the **hstr** function — are always declared using the “=” character.)

To find the index for a particular string in the language file, add a “@MEX” prefix before the definition of the string in **english.mad**. When MAID writes out the **english.lh** file, it will generate a **#define** which automatically calls the **lstr** function using the appropriate string number.

---

---

## ltostr

---

---

**Prototype** string ltostr(long: l);

**Arguments** *l* The long integer to be converted.

**Return Value** This function returns the string representation of the long integer.

**Description** The **ltostr** function is used to convert a long integer to a string. The converted string contains the ASCII representation of the long integer, from -2147483648 to 2147483647. See the **ultostr** function for converting unsigned longs.

This function is useful when strings must be mixed with the results of integral computations.

**Example** See the description for the **itostr** function for a related example.

---



---

**mdm\_command**


---



---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int mdm_command(string: cmdstring);</code>   |
| <b>Arguments</b>    | <i>cmdstring</i> The command to be sent to the modem. This string uses all of the same translation characters as in the modem initialization and busy strings from the Maximus control files, such as “ ” for <enter> and “~” for a one-second pause. Please see the <b>Busy</b> keyword in section 18.2.2 for more information. |
| <b>Return Value</b> | TRUE if the string was transmitted successfully; FALSE otherwise.  |
| <b>Description</b>  | The <b>mdm_command</b> function transmits a command string to the modem. This function is normally only used when talking directly to the modem, rather than when a user is on-line.   |

---



---

**mdm\_flow**


---



---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>void mdm_flow(int: state);</code>  |
| <b>Arguments</b>    | <i>state</i> A flag describing the desired state of XON/XOFF flow control. If this flag is set to 1, XON/XOFF flow control will be enabled if the SysOp has enabled <b>Mask Handshaking XON</b> . If this flag is set to 0, XON/XOFF flow control will always be disabled. |
| <b>Return Value</b> | None   |
| <b>Description</b>  | This function enables or disables software handshaking. Software handshaking normally needs to be disabled before trying to communicate directly with the modem.   |

---



---

**menu\_cmd**


---



---

|                  |  |
|------------------|--|
| <b>Prototype</b> | <code>void menu_cmd(int: cmdnum, string: args);</code>   |
| <b>Arguments</b> | <p><i>cmdnum</i> The <i>MNU_*</i> constant describing the menu option to execute. The <b>max_menu.mh</b> header file must be included (with the <b>#include</b> directive) to define the <i>MNU_*</i> constants.</p> <p><i>args</i> Arguments for the menu command. These arguments are specified in string format. Most menu commands do not require arguments; the only functions which require arguments are those which have an argument specified in the second</p> |



column in the menus control file. For all other menu option types, this string should be the null string ("").

**Return Value** None

**Description** The **menu\_cmd** function executes an internal Maximus menu command. Sample actions include entering a message, performing a new files search, and invoking most of the other commands in **menus.ctl**.

The main restrictions for **menu\_cmd** are:

- The **Display\_File** menu command cannot be invoked. (Instead, see the **display\_file** MEX function.)
- The **Edit\_\*** menu commands cannot be invoked unless the user is already running either the MaxEd or the BORED editor.
- The **MEX**, **Xtern\_Erlvl**, **Link\_Menu**, **Return** and **Display\_Menu** menu commands cannot be invoked.

---

---

### msg\_area

---

---

**Prototype** void msg\_area();

**Arguments** None

**Return Value** None

**Description** This function displays the message area menu. Calling **msg\_area** is equivalent to calling *menu\_cmd(MNU\_MSG\_AREA, "")*, although calling **msg\_area** is slightly faster.

---

---

### msgareafindclose

---

---

**Prototype** void msgareafindclose();

**Arguments** None

**Return Value** None

**Description** The **msgareafindclose** function terminates an existing **msgareafindfirst** search. This function should be called whenever the program has finished using the message area data file.

---

## msgareafindfirst

---

**Prototype** int msgareafindfirst(**ref** struct \_marea: ma, string: name, int: flags);

**Arguments**

*ma* A reference to a message area information structure. If this function finds a message area matching the *name* and *flags* parameters, information about that message area will be placed in this structure.

*name* Name of a specific message area to find. If *name* is the null string (""), this function will find the first available message area.

*flags* A flag indicating whether or not **MsgDivisionBegin** and **MsgDivisionEnd** records are to be returned. If this flag is equal to *AFFO\_DIV*, division records will be returned. Otherwise, if this flag is equal to *AFFO\_NODIV*, division records will be skipped.

**Return Value** TRUE if an area was found; FALSE otherwise.

**Description** The **msgareafindfirst** function searches for a specific message area, as specified by the *name* parameter. It also finds division records within the area file if the *AFFO\_DIV* flag is specified.

Only areas which can be accessed by the user will be returned by this function. This function will also skip message areas that have the “Hidden” style.

**Example** The following code will display a list of all message areas:

```
#include <max.mh>

int main()
{
    struct _marea: ma;

    if (msgareafindfirst (ma, "", AFFO_NODIV))
    {
        do
        {
            print("Area: ", ma.name, '\n');
        }
        while (msgareafindnext (ma));

        msgareafindclose();
    }
}
```

```

    }
    return 0;
}

```

---

## msgareafindnext

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int msgareafindnext(ref struct _marea: ma);</code>  |
| <b>Arguments</b>    | <i>ma</i> A reference to the message area structure to be updated with message area information. This structure must have been originally filled in by a <b>msgareafindfirst</b> , <b>msgareafindnext</b> , or <b>msgareafindprev</b> call.   |
| <b>Return Value</b> | TRUE if another message area was found; FALSE if no message area could be found.  |
| <b>Description</b>  | The <b>msgareafindnext</b> function finds the next message area that is accessible to the user. The search is carried out from the position where the last <b>msgareafindnext</b> , <b>msgareafindprev</b> , or <b>msgareafindfirst</b> call terminated. Consequently, if the last <b>msgareafind*</b> call returned information for area “X”, this function would return information for the next user-accessible area following area “X”. |

---

## msgareafindprev

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int msgareafindprev(ref struct _marea: ma);</code>   |
| <b>Arguments</b>    | <i>ma</i> A reference to the message area structure to be updated with message area information. This structure must have been originally filled in by a <b>msgareafindfirst</b> , <b>msgareafindnext</b> , or <b>msgareafindprev</b> call.  |
| <b>Return Value</b> | TRUE if another message area was found; FALSE if no message area could be found.   |
| <b>Description</b>  | The <b>msgareafindnext</b> function finds the previous message area that is accessible to the user. The search is carried out from the position the last <b>msgareafindnext</b> , <b>msgareafindprev</b> , or <b>msgareafindfirst</b> call terminated. Consequently, if the last <b>msgareafind*</b> call returned information for area “X”, this function would return information for the first user-accessible area that precedes area “X”. |

---

## msgareaselect

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int msgareaselect(string: name);   |
| <b>Arguments</b>    | <i>name</i> Name of the message area to be selected as the user's current message area   |
| <b>Return Value</b> | TRUE if the area was successfully selected; FALSE otherwise.   |
| <b>Description</b>  | The <b>msgareaselect</b> function is used to change the user's current message area. Upon return, this function also updates the global <i>mare</i> and <i>msg</i> structures with information about the new message area. |

---

## open

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int open(string: name, int: mode);   |
| <b>Arguments</b>    | <i>name</i> The name of the file to be opened.<br><br><b>WARNING!</b><br><br>All backslashes must be escaped in MEX programs. For example, to specify a file called <code>\max\misc\foo.bbs</code> , the parameter must contain <code>"\\max\\misc\\foo.bbs"</code> .<br><br><i>mode</i> One or more <i>IOPEN_*</i> constants specifying the mode to be used when opening the file. The <i>bitwise or</i> operator can be used to combine multiple <i>IOPEN_*</i> constants. This parameter is described below in more detail. |
| <b>Return Value</b> | -1 if an error occurs; otherwise, <i>open</i> returns an integer which identifies the opened file. This identifier must be stored and passed to the other MEX I/O functions when the file is to be accessed.   |
| <b>Description</b>  | The <b>open</b> function opens a file. Depending on the value of <i>mode</i> , this function can be used to open an existing file or create a new one.<br><br>The <i>mode</i> parameter indicates the file-opening mode. <i>mode</i> should contain exactly one of the constants from Table 15.38.   |

**Table 15.38 File Open Modes**

| Mode        | Description               |
|-------------|---------------------------|
| IOPEN_READ  | Open the file for reading |
| IOPEN_WRITE | Open the file for writing |

In addition, any of the modifiers from Table 15.39 can also be used if IO-PEN\_WRITE is specified:

Table 15.39 File Open Modifiers

| Mode         | Description   |
|--------------|---|
| IOPEN_APPEND | Append to the end of the file.  |
| IOPEN_CREATE | Create the file if it does not exist, or truncate the file if it does.                          |
| IOPEN_BINARY | Open the file in binary mode. This option suppresses the translation of end-of-line characters. |

**Example** The following code creates a file called **c:\test.fil** and makes the file ready for writing:

```
int: fd;

fd := open("c:\\test.fil",
           IOPEN_CREATE | IOPEN_WRITE);
```

---

## print

---

**Prototype** void print(...);

**Arguments** Any number of parameters (with any type) can be specified. See the function description for more information.

**Return Value** None

**Description** **print** is used to display text to the user. **print** can display any type of information, including characters, integers, strings, or even user-defined structures or data types.

Any number of parameters may be specified to **print**, and all will be displayed using formatting routines appropriate to the data type.

The **print** function displays the information specified in its parameters, but it does not automatically place the cursor on the next line. To add this functionality, include a ‘\n’ parameter at the end of the function call.

**print** handles the data types shown in Table 15.40 by default:

Table 15.40 Print Data Types

| Type          | Description  |
|---------------|--|
| char          | Display a character in its natural format. For example, the character ‘A’ will be displayed simply as “A.” Control characters and high-bit characters will also be displayed as-is, although all print output will pass through the standard Maximum output filter. This means that high-bit characters may sometimes be replaced with ASCII equivalents, and terminal control sequences may be stripped. (See section 13 for more information.) |
| int           | Display an integer in decimal. The range for signed integers is minus 32768 to 32767.  |
| long          | Display a long integer in decimal. The range for longs is minus 2147483648 to 2147483647.  |
| unsigned int  | Display an unsigned integer in decimal. The range for unsigned integers is 0 to 65535.   |
| unsigned long | Display an unsigned long in decimal. The range for unsigned longs is 0 to 4294967296.  |
| string        | Display a string. The string will be displayed just as if each of the characters inside had been displayed individually as a char, as above.   |

**print** itself is just a meta-function — there is no real function called “print” in the MEX run-time library. However, when the compiler encounters a **print** statement, it splits apart all of the arguments and calls a separate function for each.

For an argument with a type of *mytype*, the MEX compiler will generate a call to a function of the form:

```
__printMYTYPE(data);
```

where *MYTYPE* is the uppercase name of the type to be displayed. The standard run-time library includes the following **print** handlers:

```
__printSTRING
__printLONG
__printINT
__printCHAR
__printUNSIGNED_LONG
__printUNSIGNED_INT
__printUNSIGNED_CHAR
```

Support for user-defined data types can also be added to **print** by defining a new **print** function to handle the appropriate argument type. For example, given a data type and an appropriate **print** handler function:

```
struct complex
```

```

{
    int: real;
    int: imaginary;
};

void __printSTRUCT_COMPLEX(ref struct complex: c)
{
    print('(' , c.real , ',' , c.imaginary , ')');
}

```

With these definitions, the **print** function can be used to print a structure of type *complex* by simply passing the structure as a parameter:

```

struct complex: c;

c.real := 5;
c.imaginary := 10;

print("The complex number is ", c, ".\n");

```

---

## privok

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int privok(string: acs);  |
| <b>Arguments</b>    | <i>acs</i> The Access Control String (ACS) to be checked.   |
| <b>Return Value</b> | TRUE if the user's privilege level passes the privilege level check; FALSE otherwise.   |
| <b>Description</b>  | The <b>privok</b> function is used to check a user's privilege level against a given Access Control String. The ACS check performed by this function is the same as in all other areas of Maximus, so all of the standard ACS modifiers (">=", "!", and so on) can be specified in <i>acs</i> . |

---

## prm\_string

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | string prm_string(int: stringnum);   |
| <b>Arguments</b>    | <i>stringnum</i> The string number to be retrieved from the Maximus <b>.prm</b> file. A string number of 0 specifies the first string. |
| <b>Return Value</b> | The string that was retrieved, or the null string ("") if the string number is invalid.  |

**Description** This function retrieves a string from the Maximus **.prm** file. The string number specifies an offset within the fixed-length string index table. The string numbers are version-specific and are subject to change without notice.

---

### protocol\_num\_to\_name

---

**Prototype** string protocol\_num\_to\_name(int: protocol);

**Arguments** *protocol* An integer index for a protocol defined in *protocol.ctl*.

**Return Value** A string containing the name of the protocol, or the null string ("") if the protocol index is invalid.

**Description** This function is used to translate the *usr.def\_proto* field in the user structure to obtain a human-readable string.

---

### read

---

**Prototype** int read(int: fd, **ref** string: s, int: len);

**Arguments** *fd* A file descriptor, as returned by the **open** function.

*s* A reference to a string. Upon return, this string is filled in with the bytes read from the file.

*len* The maximum number of bytes to place into the string *s*.

**Return Value** If the return value is the same as *len*, the read was completely successful.

If the return value is less than *len* (but greater than zero), only a portion of the requested number of bytes could be read.

If the return value is 0, end-of-file was encountered.

If the return value is -1, an error occurred when trying to read from the file.

**Description** The **read** function reads a block of bytes from the specified file handle. This function reads blocks of bytes at a time with no consideration for “lines” in the source file. To read a file a line at a time, see the **readln** function.



---



---

## readln

---



---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int readln(int: fd, ref string: s);</code>   |
| <b>Arguments</b>    | <p><i>fd</i>      A file descriptor, as returned by the <b>open</b> function.</p> <p><i>s</i>      A reference to a string. Upon return, this string will be filled in with the line read from the file.</p>   |
| <b>Return Value</b> | <p>If the return value is greater than zero, this is the number of bytes placed into the string.</p> <p>If the return value is 0, end-of-file was encountered</p> <p>If the return value is -1, an error occurred when trying to read from the file.</p> |
| <b>Description</b>  | The <b>readln</b> function reads an entire line from the specified file handle. If the line ends with a newline character, it is automatically stripped.   |

---



---

## remove

---



---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int remove(string: file);</code>   |
| <b>Arguments</b>    | <p><i>file</i>      The name of the file to be deleted.</p> <p><b>WARNING!</b></p> <p>All backslashes must be escaped in MEX programs. For example, to specify a file called <code>\max\misc\foo.bbs</code>, the parameter must contain <code>"\\max\\misc\\foo.bbs"</code>.</p> |
| <b>Return Value</b> | TRUE if the file was successfully deleted; FALSE otherwise.  |
| <b>Description</b>  | This function deletes the specified filename.  |

---



---

## rename

---



---

|                  |  |
|------------------|--|
| <b>Prototype</b> | <code>int rename(string: old, string: new);</code>         |
| <b>Arguments</b> | <p><i>old</i>      The name of the file to be renamed.</p> |

**WARNING!**

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain `"\\max\\misc\\foo.bbs"`.

*new*      The new name to be assigned to the file.

**Return Value**      TRUE if the rename operation was successful; FALSE otherwise.

**Description**      The **rename** function is used to rename or move an existing file. If the file is not in the current directory, full paths must be specified for both *old* and *new*.

This function cannot be used to move files across drives.

---

**reset\_more**


---

**Prototype**      void reset\_more(ref char: nonstop);

**Arguments**      *nonstop*    A reference to the non-stop control character. This character is initialized to a value of 0 by **reset\_more**.

**Return Value**      None

**Description**      The **reset\_more** function resets the internal “More” counter that controls the number of screen lines remaining until a “More [Y,n,=]?” prompt is displayed. The current point in the display will be treated as the “top” of the current output page, insofar as more prompts are concerned.

The *nonstop* variable is used in later calls to the **do\_more** function (which is where the more prompts are actually displayed).

---

**rip\_detect**


---

**Prototype**      int rip\_detect();

**Arguments**      None

**Return Value**      TRUE if the user’s terminal supports RIP<sub>scrip</sub> graphics; FALSE otherwise.

**Description** The **rip\_detect** function is used to query the remote terminal to determine if it supports *RIPscrip* graphics. If the remote terminal reports that *RIPscrip* graphics are supported, this function returns TRUE.

After the user has logged on, the user's current preference for *RIPscrip* graphics can be read from the *usr.rip* field.

---

## rip\_hasfile

---

**Prototype** int rip\_hasfile(string: fname, **ref** long: filesize);

**Arguments** *fname* The name of the remote file to query. This name must *not* have an explicit path.

*filesize* A reference to the size of the file to be queried. If *filesize* is set to -1, Maximus will query the remote for the size of the file and place it into this variable upon return.

**Return Value** 1 if the remote user has the file;  
0 if the remote user does not have the file;  
-1 if a *RIPscrip* protocol error occurred

**Description** The **rip\_hasfile** function allows a MEX program to determine whether or not the remote user has a specified *RIPscrip* file.

If an explicit *filesize* is provided, this function only returns TRUE if the remote user has the file and the file has the indicated size.

Otherwise, if *filesize* is set to -1 before calling **rip\_hasfile**, Maximus checks the remote side and sets the *filesize* parameter to the size of the remote file. It returns TRUE if the file exists and FALSE otherwise.

---

## rip\_send

---

**Prototype** int rip\_send(string: filename, int: display);

**Arguments** *filename* The filename to be sent to the remote *RIPscrip* user. If no path is specified, Maximus will assume the current **RIP Path**.

*display* TRUE if the file is to be displayed as soon as it is sent; FALSE otherwise.

**Return Value** TRUE if the file was successfully displayed/sent; FALSE otherwise.

**Description** This function is used to send a *RIPscrip* scene or icon file to the remote user. After sending the file, it can be optionally displayed by setting the *display* parameter to TRUE.

This performs a function equivalent to the *[ripsend]* MECCA token.

---

---

## screen\_length

---

---

**Prototype** int screen\_length();

**Arguments** None

**Return Value** The length of the local screen, in rows.

**Description** The **screen\_length** function returns the length of the local console screen.

---

---

## screen\_width

---

---

**Prototype** int screen\_width();

**Arguments** None

**Return Value** The width of the local screen, in columns.

**Description** The **screen\_width** function returns the width of the local console screen.

---

---

## seek

---

---

**Prototype** int seek(int: fd, long: pos, int: where);

**Arguments**

*fd* A file descriptor, as returned by the **open** function.

*pos* The position to which the file should be seeked. This position is relative to the offset specified for the *where* parameter. Negative offsets are permitted if either *SEEK\_CUR* or *SEEK\_END* are specified for *where*.

*where* This parameter defines the relation between the *pos* parameter and the physical offset within the file. This is described in more detail below.

- Return Value** The new file offset, relative to the beginning of the file.
- Description** The **seek** function moves the file pointer for the specified file to a new location. This function is used to jump to an arbitrary offset within a file, or to jump directly to the beginning or end of a file.

The *where* parameter must be one of the values from Table 15.41:

**Table 15.41 Seek Offsets**

| Value    | Description  |
|----------|--|
| SEEK_CUR | <i>pos</i> is relative to the current file position. |
| SEEK_SET | <i>pos</i> is relative to the beginning of the file. |
| SEEK_END | <i>pos</i> is relative to the end of the file.       |

---

## set\_output

---

- Prototype** `int set_output(int mode);`
- Arguments** *mode* This function sets the output control mode. This must be one of the *DISABLE\_\** parameters described below.
- Return Value** The original setting for screen output. This return value can be used to restore the screen output mode at a later time by calling **set\_output** again.
- Description** This function allows the Maximus video output to be suppressed, for either the remote user, the local screen, or both.

The *mode* parameter must be one of the values from Table 15.42:

**Table 15.42 Output Disable Modes**

| Value          | Description                           |
|----------------|---------------------------------------|
| DISABLE_NONE   | Enable all output.                    |
| DISABLE_LOCAL  | Disable only local output.            |
| DISABLE_REMOTE | Disable only remote output.           |
| DISABLE_BOTH   | Disable both local and remote output. |

---

## set\_static\_data

---

- Prototype** `int set_static_data(string: key, ref void: data);`

|                     |  |
|---------------------|--|
| <b>Arguments</b>    | <p><i>key</i>      The key for the static data object to be set. This must be the same as the <i>key</i> parameter passed to <b>create_static_data</b>.</p> <p><i>data</i>      A reference to the local data object to store in the static data object. Note that this parameter is a <i>void reference</i>, meaning that any type of object (char, int, long, array or struct) can be referenced.</p>  |
| <b>Return Value</b> | <p>0 if the data object was stored successfully;<br/>-1 if the key name was not found.</p>   |
| <b>Description</b>  | <p>The <b>set_static_data</b> function is used to store data that can be retrieved later during the same Maximus session by the <b>get_static_data</b> function. Data stored by <b>set_static_data</b> is persistent, in that it retains its value even after the creating MEX program has ended.</p> <p>If <i>key</i> is a valid static data key that was created by <b>create_static_data</b>, this function will copy the information from the local structure referenced by <i>data</i> into the static data object.</p> |

---

## set\_static\_string

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int set_static_string(string: key, string: data);   |
| <b>Arguments</b>    | <p><i>key</i>      The key for the static string to be set. This must be the same as the <i>key</i> parameter passed to <b>create_static_string</b>.</p> <p><i>data</i>      The local string to be stored in the static string.</p>  |
| <b>Return Value</b> | <p>0 if the string was stored successfully;<br/>-1 if the key name was not found;<br/>-2 if there was not enough memory to store the string.</p>  |
| <b>Description</b>  | <p>The <b>set_static_string</b> function is used to store strings that can be retrieved later during the same Maximus session by the <b>get_static_string</b> function. Strings stored by <b>set_static_string</b> are persistent, in that they retain their values even after the creating MEX program has ended.</p> <p>If <i>key</i> is a valid static string key that was created by <b>create_static_string</b>, this function will copy the information from the local string referenced by <i>data</i> into the static string.</p> |

---

**set\_textsize**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | void set_textsize(int: cols, int: rows);  |
| <b>Arguments</b>    | <p><i>cols</i>      The number of columns in the window.</p> <p><i>rows</i>      The number of rows in the window.</p>  |
| <b>Return Value</b> | None  |
| <b>Description</b>  | <p>The <b>set_textsize</b> sets the assumed text window size of the remote system. This is primarily used to set the size of the display for “More” prompting when using <i>RIPscrip</i> windows on the remote terminal.</p> <p>Specifying values of 0 for either <i>cols</i> or <i>rows</i> will set the window width or length (respectively) back to the default, as specified in the user record.</p> |

---

**shell**


---

|                  |   |
|------------------|---|
| <b>Prototype</b> | int shell(int: method, string: cmd);  |
| <b>Arguments</b> | <p><i>method</i>    An <i>IOUTSIDE_*</i> constant describing the method to use for executing the program. This parameter is described below in more detail.</p> <p><i>cmd</i>        The name of the command to run, plus any optional program arguments.</p> |

**WARNING!**

All backslashes must be escaped in MEX programs. For example, to specify a file called `\max\misc\foo.bbs`, the parameter must contain “`\\max\\misc\\foo.bbs`”.

|                     |  |
|---------------------|--|
| <b>Return Value</b> | The return value of the program, or -1 if the program could not be executed. |
|---------------------|--|

|                    |   |
|--------------------|---|
| <b>Description</b> | The <b>shell</b> function invokes an external program or a secondary copy of the command interpreter. The exact method used for invoking the external program depends on the value of the <i>method</i> parameter. At least one of <i>IOUTSIDE_RUN</i> or <i>IOUTSIDE_DOS</i> must be specified; the other parameters from Table 15.43 are optional and can be combined using the <i>bitwise or</i> operator: |
|--------------------|---|

**Table 15.43 Outside Methods**

| Value               | Description   |
|---------------------|---|
| <i>IOUTSIDE_RUN</i> | Spawn the program directly. This option can only be |

|                 |  |
|-----------------|--|
|                 | used to run .exe and .com files in the operating system's native format. This method is faster than IOUTSIDE_DOS.  |
| IOUTSIDE_DOS    | Execute the program through the command interpreter. This option can be used to spawn .bat, .cmd, .exe and .com files, in addition to internal shell commands (such as "dir" and "copy"). Under OS/2, this function can also be used to invoke DOS programs. |
| IOUTSIDE_REREAD | After running the external program, re-read the user record from the lastuser.bbs file. This flag can be combined with the IOUTSIDE_RUN or IOUTSIDE_DOS flags using the bitwise or operator.   |

---

**Example**

The following code displays a directory of the \MAX\MISC directory:

```
shell(IOUTSIDE_DOS, "dir c:\\max\\misc");
```

---

## sleep

---

**Prototype**

```
void sleep(int: duration);
```

**Arguments**

*duration* The amount of time to sleep, measured in hundredths of seconds.

**Return Value**

None

**Description**

The **sleep** function instructs Maximus to pause for a certain period of time. Since *duration* is measured in hundredths of seconds, a value of 500 would tell Maximus to sleep for five seconds.

Under OS/2, this function can be used to temporarily yield control to other programs during a polling loop. Calling *sleep(1)* tells Maximus to yield for long enough for other programs to run, but also to return control to the MEX program quickly enough so that there is no noticeable lag in response time.

---

## snoop

---

**Prototype**

```
int snoop(int: state);
```

**Arguments**

*state* The new state for the console snoop mode. A value of TRUE enables snoop mode, while a value of FALSE disables snoop mode.



**Return Value** The original setting of snoop mode. This value can be used at a later time to restore the original snoop mode setting.

**Description** The **snoop** function is used to set the state of the internal “snoop” feature. When snoop is enabled, the local console will show exactly what is displayed on the remote screen.

---

## stamp\_string

---

**Prototype** string stamp\_string(**ref** struct \_stamp: t);

**Arguments** *t* A reference to a structure containing date and time values.

**Return Value** A string version of the date and time, using the format specified in *max.ctl*.

**Description** The **stamp\_string** function converts a *\_stamp* structure into a human-readable string, using the SysOp-defined time format in the **max.ctl** file.

---

## stamp\_to\_long

---

**Prototype** long stamp\_to\_long(**ref** struct \_stamp: st);

**Arguments** *st* A reference to a structure containing date and time values.

**Return Value** A long integer representing the number of seconds elapsed since January 1st, 1970 UTC.

**Description** The **stamp\_to\_long** function converts a *\_stamp* structure back into the format that is returned by the **time** function. This function is useful when the date/time entries into two *\_stamp* structures need to be compared in terms of chronological order.

---

## strfind

---

**Prototype** int strfind(**ref** string: str, string: substring);

**Arguments** *string* The string to be searched.

*substring* The substring to search for within *string*.

- Return Value** 0 if *substring* was not found; otherwise, the index in *string* at which *substring* is found. An index of 1 indicates the first byte in *string*.
- Description** The **strfind** function tries to find an occurrence of *substring* within the master string *str*.
- Example** This shows how the **strfind** return value is used:

```
#include <max.mh>

int main()
{
    int: i;

    i := strfind("This is a big string", "big");
    print(i);    // i = 11
    return 0;
}
```

---

## stridx

---

- Prototype** int stridx(string: src, int: startpos, int: ch);
- Arguments**
- src*            The string to be searched.
  - startpos*    The position in *src* at which the search is to start.
  - ch*            The character to search for within *src*.
- Return Value** 0 if the character could not be found; otherwise, the index of the position containing the character.
- Description** The **stridx** function searches the string specified by *src* for any occurrences of the character *ch*. If *ch* is found, it returns the index of that character within the string.
- This function searches the string from left to right, starting at the position specified. To search the string from right to left, see the **strridx** function.
- Example** To search for all instances of a character within a given string, the following code can be used:

```
#include <max.mh>

int main()
{
    int: pos;
```

```

string: src;

src := "Abcdxefghxijklmnoxpqxrxxst";
pos := 1;

for (pos := stridx (src, pos, 'x');
    pos;
    pos := stridx (src, pos+1, 'x'))
{
    print("Found an 'x' at position ",
        pos, '\n');
}

return 0;
}

```

---

## strlen

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int strlen(string: s);   |
| <b>Arguments</b>    | <i>s</i> The string to be measured.  |
| <b>Return Value</b> | The length of the string.  |
| <b>Description</b>  | <p>The <b>strlen</b> function determines the length of a given string and returns it to the caller.</p> <p>An empty or uninitialized string has a length of 0.</p> |

---

## strlower

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string strlower(string: src);                 |
| <b>Arguments</b>    | <i>src</i> The string to be converted.        |
| <b>Return Value</b> | A lowercase version of the <i>src</i> string. |
| <b>Description</b>  | This function converts a string to lowercase. |

---

## strpad

---

|                  |   |
|------------------|---|
| <b>Prototype</b> | string strpad(string: str, int: length, char: pad); |
|------------------|---|

|                     |   |  |
|---------------------|---|--|
| <b>Arguments</b>    | <i>str</i>  | The string to be padded                                    |
|                     | <i>length</i>   | The length to which the string should be padded.           |
|                     | <i>pad</i>  | The character which should be used for padding the string. |
| <b>Return Value</b> | The padded version of the string.   |  |
| <b>Description</b>  | The <b>strpad</b> function pads a string so that it is at least <i>length</i> characters long.  |  |
|                     | If the string is already more than or equal to <i>length</i> characters in length, the string is returned unchanged.  |  |
|                     | If the string is less than <i>length</i> characters in length, the <i>pad</i> character is appended to the end of the string as many times as necessary to make the string exactly <i>length</i> characters long. |  |

---



---

## strpadleft

---



---

|                     |   |  |
|---------------------|---|--|
| <b>Prototype</b>    | string strpadleft(string: str, int: length, char: pad);   |  |
| <b>Arguments</b>    | <i>str</i>  | The string to be padded                                    |
|                     | <i>length</i>   | The length to which the string should be padded.           |
|                     | <i>pad</i>  | The character which should be used for padding the string. |
| <b>Return Value</b> | The padded version of the string.   |  |
| <b>Description</b>  | The <b>strpadleft</b> function pads a string so that it is at least <i>length</i> characters long.  |  |
|                     | This function differs from <b>strpad</b> only in that the padding characters are added to the beginning of the string rather than at the end of the string. |  |

---



---

## stridx

---



---

|                  |  |   |
|------------------|--|---|
| <b>Prototype</b> | int stridx(string: src, int: startpos, int: ch); |   |
| <b>Arguments</b> | <i>src</i>                                       | The string to be searched.  |
|                  | <i>startpos</i>                                  | The position in <i>src</i> at which the search is to start. A value of 0 for <i>startpos</i> instructs Maximus to start searching from the end of the string. |

*ch*            The character to search for within *src*.

**Return Value**    0 if the character could not be found; otherwise, the index of the position containing the character.

**Description**    The **strridx** function searches the string specified by *src* for any occurrences of the character *ch*. If *ch* is found, it returns the index of that character within the string.

This function searches the string from right to left, starting at the position specified, or at the end of the string if *startpos* is 0. To search the string from left to right, see the **stridx** function.

**Example**            To search for all instances of a character within a given string, the following code can be used. (The character indices are returned in order from right to left.)

```
#include <max.mh>

int main()
{
    string: src;
    int: pos;

    src := "Abcdxefgxiijklmxxxopqrxrstux";
    pos := 0;

    for (pos := strridx(src, pos, 'x');
        pos;
        pos := strridx(src, pos-1, 'x'))
    {
        print("Found an 'x' at position ",
            pos, '\n');
    }

    return 0;
}
```

---

---

## strtoi

---

---

**Prototype**        int strtoi(string: s);

**Arguments**        *s*            The string containing the decimal representation of a number.

**Return Value**    The integer equivalent of the string, or 0 if the string could not be converted.

**Description**        The **strtoi** function converts an ASCII string into an integer. The string must have a digit as its first character, and the number represented in the string must be in decimal.

This function reads the string until it encounters a non-digit or the end of the string. All of the digits up to that point are converted and returned to the caller as an integer.

This function can handle numbers in the range of -32768 through 65535. However, the exact range of usable numbers depends on the type of integer to which the return code is assigned.

If the return code is assigned to an unsigned integer, values in the range 0 to 65535 are converted. Otherwise, if the return code is assigned to a signed integer, values in the range -32768 to 32767 are converted. To handle larger numbers, see the **strtol** function.

---

## strtok

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int strtok(string: src, string: toks, <b>ref</b> int: pos);</code>  |
| <b>Arguments</b>    | <p><i>src</i>            The source string to be tokenized.</p> <p><i>toks</i>           A string containing a list of acceptable token delimiters. These token delimiters are used to delimit where one token ends and the following token begins. The end of the string is always considered to be a delimiter.</p> <p><i>pos</i>            A reference to an integer containing the most-recently examined position in the string. On the initial call to <b>strtok</b>, this value should be set to 0. This variable is normally updated by <b>strtok</b>, so applications should not need to modify <i>pos</i> after the first call to <b>strtok</b>.</p> |
| <b>Return Value</b> | A string containing the token, stripped of all token delimiters. If no more tokens exist in the string, the null string ("") is returned.   |
| <b>Description</b>  | The <b>strtok</b> function is used for parsing strings. It returns the substring in <i>src</i> , starting at position <i>pos</i> , which is delimited by any of the characters in the <i>toks</i> string.   |
| <b>Example</b>      | <p>The following code tokenizes a string, using spaces and tabs as delimiters:</p> <pre>#include &lt;max.mh&gt;  int main() {     string: src, sub;     int: pos;      pos := 0;</pre>  |

```

src := "This is a test";

for (sub := strtok (src, " \t", pos);
    sub <> "";
    sub := strtok (src, " \t", pos))
{
    print("Word is '", sub, "'\n");
}

// The program prints:
//
// Word is 'This'
// Word is 'is'
// Word is 'a'
// Word is 'test'

return 0;
}

```

---

## strtol

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int strtol(string: s);   |
| <b>Arguments</b>    | <i>s</i> The string containing the decimal representation of a number.   |
| <b>Return Value</b> | The long integer equivalent of the string, or 0 if the string could not be converted.  |
| <b>Description</b>  | <p>The <b>strtol</b> function converts an ASCII string into a long integer. The string must have a digit as its first character, and the number represented in the string must be in decimal.</p> <p>This function reads the string until it encounters a non-digit or the end of the string. All of the digits up to that point are converted and returned to the caller.</p> <p>This function can handle numbers in the range of -2147483648 to 4294967296. However, the exact range of usable numbers depends on the type of long integer to which the return code is assigned.</p> <p>If the return code is assigned to an unsigned long, values in the range 0 to 4294967296 are converted. Otherwise, if the return code is assigned to a signed integer, values in the range -2147483648 to 2147483647 are converted.</p> |

---

**strtrim**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string strtrim(string: src, string: chrs);  |
| <b>Arguments</b>    | <p><i>src</i>            The string to be trimmed.</p> <p><i>chrs</i>           A string containing a list of characters to be trimmed from the beginning and end of <i>src</i>.</p>  |
| <b>Return Value</b> | The trimmed string.   |
| <b>Description</b>  | The <b>strtrim</b> function adjusts a string to remove leading and trailing characters specified in the <i>chrs</i> string. The returned string contains a copy of the original string with the leading and trailing characters stripped. |
| <b>Example</b>      | This code demonstrates the usage of the <b>strtrim</b> function:  |

```
string: trimmed;

trimmed := strtrim("What to do now?", "Wh?");
// trimmed contains "at to do now"
```

---

**strupper**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | string strupper(string: src);                  |
| <b>Arguments</b>    | <i>src</i> The string to be converted.         |
| <b>Return Value</b> | An uppercase version of the <i>src</i> string. |
| <b>Description</b>  | This function converts a string to uppercase.  |

---

**substr**


---

|                  |  |
|------------------|--|
| <b>Prototype</b> | string substr(string: s, int: pos, int: length);   |
| <b>Arguments</b> | <p><i>s</i>                The source string from which the substring is to be extracted.</p> <p><i>pos</i>             The starting position within <i>s</i> from which the substring is to be taken. A value of 1 specifies the first character in <i>s</i>.</p> |



*length* The maximum length of the substring to be extracted from *s*. The **substr** function will normally extract exactly *length* characters, but if this character count would exceed the length of the string, fewer characters will be extracted.

**Return Value** The substring extracted from *s*.

**Description** The **substr** function extracts a substring from the source string *s*. The substring is defined by specifying a starting position within the string, and also by specifying the number of following characters which are to be taken as part of that substring.

**Example** This code shows how the **substr** function operates:

```
#include <max.mh>

int main()
{
    string: str;

    str := substr("The quick brown fox", 11, 5);

    print(str); // str now contains "brown"
    return 0;
}
```

---

---

## tag\_dequeue\_file

---

---

**Prototype** int tag\_dequeue\_file(int: posn);

**Arguments** *posn* Position of the file within the tag list. The first file in the list is position 0.

**Return Value** TRUE if the file was successfully dequeued; FALSE otherwise.

**Description** The **tag\_dequeue\_file** function is used to remove files from the internal queue of files to be downloaded.

The number of files in the queue can be obtained using the **tag\_queue\_size** function, and information about specific queue entries can be obtained using the **tag\_get\_name** function.

---

---

## tag\_get\_name

---

---

**Prototype** int tag\_get\_name(int: posn, **ref** int: flags, **ref** string: filename);

|                     |  |   |
|---------------------|--|---|
| <b>Arguments</b>    | <i>posn</i>  | Position of the file within the tag list. The first file in the list is position 0.   |
|                     | <i>flags</i>   | Upon return, this variable is updated with a copy of the flags indicating the attributes for the file. These attributes are described in more detail in the <b>tag_queue_file</b> function. |
|                     | <i>filename</i>  | A reference to a string. Upon return, this string is updated with the full filename and path of the file in the specified queue position.   |
| <b>Return Value</b> | TRUE if the file information was successfully queried; FALSE otherwise.  |   |
| <b>Description</b>  | The <b>tag_get_name</b> function is used to determine the names and attributes of files in the download queue. |   |

---

## tag\_queue\_file

---

|                     |  |  |
|---------------------|--|--|
| <b>Prototype</b>    | int tag_queue_file(string: filename, int: flags);  |  |
| <b>Arguments</b>    | <i>filename</i>  | The full path and filename of the file to be added to the queue.   |
|                     | <p><b>WARNING!</b></p> <p>All backslashes must be escaped in MEX programs. For example, to specify a file called <code>\max\misc\foo.bbs</code>, the parameter must contain <code>"\\max\\misc\\foo.bbs"</code>.</p>                               |  |
|                     | <i>flags</i>   | A list of zero or more <i>FFLAG_*</i> file attributes to be assigned to this file. Multiple attributes can be combined using the <i>bitwise or</i> operator. |
| <b>Return Value</b> | TRUE if the file was successfully queued; FALSE otherwise.   |  |
| <b>Description</b>  | The <b>tag_queue_file</b> inserts a specific file into the user's download queue. The full path and filename of the file must be given. In addition, a number of optional flags from Table 15.44 can be set to describe the file being downloaded: |  |

**Table 15.44 File Queue Flags**

| Flag          | Description   |
|---------------|---|
| FFLAG_NOTIME  | The file is not counted against the user's time limit.  |
| FFLAG_NOBYTES | The file is not counted against the user's file download limit.   |
| FFLAG_STAGE   | The file is to be copied to the staging path before being sent to the user, as is typically done for CD-ROM drives. |

|            |   |
|------------|---|
| FFLAG_SLOW | The file is on slow media, so Maximus will try not to access the drive any more than necessary. |
|------------|---|

---

## tag\_queue\_size

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int tag_queue_size();   |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | The number of files in the download queue.  |
| <b>Description</b>  | This function returns the number of files that are currently in the download queue. The other <b>tag_*</b> functions use zero-based position numbers, so a queue containing 5 files will have file entries at positions 0, 1, 2, 3 and 4. |

---

## tell

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | long tell(int: fd);   |
| <b>Arguments</b>    | <i>fd</i> A file descriptor, as returned by the <b>open</b> function.   |
| <b>Return Value</b> | The position within the file where the next <b>read</b> , <b>readln</b> , <b>write</b> , or <b>writeln</b> will take place. |
| <b>Description</b>  | The <b>tell</b> function is used to determine the current offset of a file.   |

---

## term\_length

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | int term_length();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | The length of the user's terminal (in rows).  |
| <b>Description</b>  | The <b>term_length</b> function returns the current length of the user's terminal. If a <b>set_textsize</b> call has been made, the length parameter passed to that function is returned here. Otherwise, the value from the user record is returned. |

---

**term\_width**

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int term_width();</code>   |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | The width of the user's terminal (in columns).   |
| <b>Description</b>  | The <b>term_width</b> function returns the current width of the user's terminal. If a <b>set_textsize</b> call has been made, the width parameter passed to that function is returned here. Otherwise, the value from the user record is returned. |

---

**time**

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>long time();</code>                                    |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | The number of seconds elapsed since January 1st 1970 (UTC).  |
| <b>Description</b>  | This function returns the current value of the system timer. |

---

**time\_check**

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int time_check(int: state);</code>   |
| <b>Arguments</b>    | <i>state</i> The new state for time checking. If TRUE, time checking is enabled. Otherwise, time checking is disabled.   |
| <b>Return Value</b> | The original time checking state. This returns TRUE if the time checking was enabled prior to this function call; FALSE otherwise. This value can be used to restore the original time checking state at a later time.                   |
| <b>Description</b>  | The <b>time_check</b> function is used to enable or disable time checking. Disabling time checking effectively “freezes” the user timer countdown, which would be desirable when writing a MEX-based chat program or call-back verifier. |

---

## timeadjust

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | long timeadjust(long: delta);  |
| <b>Arguments</b>    | <i>delta</i> The number of seconds to be added to the user's current time limit. A negative value for <i>delta</i> indicates that the user's time limit is to be decreased by the indicated amount.  |
| <b>Return Value</b> | The user's new time limit, measured in seconds.  |
| <b>Description</b>  | <p>The <b>timeadjust</b> function is used to adjust the user's current time limit. Note that this function disregards the event file and the -t command line parameter. If this function is used blindly, the user could easily be allowed to overrun a scheduled event.</p> <p>To get the user's current time limit measured in seconds, call <i>timeadjust(0)</i>.</p> |

---

## timeadjustsoft

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | long timeadjustsoft(long: delta);   |
| <b>Arguments</b>    | <i>delta</i> The number of seconds to be added to the user's current time limit. A negative value for <i>delta</i> indicates that the user's time limit is to be decreased by the indicated amount. |
| <b>Return Value</b> | The user's new time limit, measured in seconds.   |
| <b>Description</b>  | <p>The <b>timeadjustsoft</b> function is used to adjust the user's current time limit. This function will not allow the user's time limit to be adjusted to exceed an external event.</p>           |

---

## timeleft

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | long timeleft();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | The user's remaining time limit, in minutes.  |
| <b>Description</b>  | <p>The <b>timeleft</b> function returns a value indicating the length of time that the current user is allowed to stay on-line, in minutes.</p> |

---

**timeon**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | long timeon();  |
| <b>Arguments</b>    | None  |
| <b>Return Value</b> | The number of minutes that the user has been logged on.   |
| <b>Description</b>  | The <b>timeon</b> function returns the amount of time elapsed for the current call, in minutes. |

---

**timestamp**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | void timestamp( <b>ref</b> struct _stamp: stamp);  |
| <b>Arguments</b>    | <i>stamp</i> A reference to a <i>_stamp</i> structure. Upon return, this structure will be updated with a copy of the current date and time. |
| <b>Return Value</b> | None   |
| <b>Description</b>  | This function retrieves the current date and time.   |

---

**uitostr**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string uitostr(unsigned int: i);  |
| <b>Arguments</b>    | <i>i</i> The unsigned integer to be converted.  |
| <b>Return Value</b> | This function returns the string representation of the unsigned integer.  |
| <b>Description</b>  | <p>The <b>itostr</b> function is used to convert an unsigned integer to a string. The converted string contains the ASCII representation of the integer, from 0 to 65535. See the <b>itostr</b> function for converting signed integers.</p> <p>This function is useful when strings must be mixed with the results of integral computations.</p> |

---

**ultostr**


---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | string ultostr(unsigned long: l);   |
| <b>Arguments</b>    | <i>l</i> The unsigned long integer to be converted.   |
| <b>Return Value</b> | This function returns the string representation of the long integer.  |
| <b>Description</b>  | <p>The <b>ultostr</b> function is used to convert an unsigned long integer to a string. The converted string contains the ASCII representation of the unsigned long integer, from 0 to 4294967295. See the <b>ltostr</b> function for converting signed longs.</p> <p>This function is useful when strings must be mixed with the results of integral computations.</p> |
| <b>Example</b>      | See the description for the <b>itostr</b> function for a related example.   |

---

**usercreate**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | int usercreate( <b>ref</b> struct _usr: u);  |
| <b>Arguments</b>    | <i>u</i> A reference to a user structure containing the user to be added.  |
| <b>Return Value</b> | TRUE if the user was successfully added; FALSE otherwise. (This function may fail if you try to add a user with a name or alias that already exists.)  |
| <b>Description</b>  | <p>The <b>usercreate</b> function adds a user to the user file. The <i>u</i> structure should be completely filled out before the <b>usercreate</b> function is called.</p> <p>This function automatically assigns a new lastread pointer to the user before writing it to the user file. (The contents of <i>u.lastread_ptr</i> are ignored.)</p> |

---

**userfilesize**


---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | long userfilesize();                                 |
| <b>Arguments</b>    | None   |
| <b>Return Value</b> | The size of the user file, measured in user records. |

**Description** The **userfilesize** returns the number of records (both deleted and non-deleted) that are present in the user file.

---

**userfindclose**

---

**Prototype** void userfindclose();

**Arguments** None

**Return Value** None

**Description** The **userfindclose** function releases the resource associated with the most recent call to **userfindopen**.

---

**userfindnext**

---

**Prototype** int userfindnext(**ref** struct \_usr: u);

**Arguments** *u* A reference to a user structure. This should be a structure returned by a previous call to **userfindopen**, **userfindnext** or **userfindprev**. Upon exit, it is updated with a copy of the new user record.

**Return Value** TRUE if the following user record was found; FALSE otherwise.

**Description** The **userfindnext** function finds the next record in the user file and returns it in the *u* structure.

---

**userfindopen**

---

**Prototype** int userfindopen(string: name, string: alias, **ref** struct \_usr: u);

**Arguments** *name* The name of a user to search for within the user file. If *name* is the null string (""), the name field in the user record is not compared.

*alias* The alias of a user to search for within the user file. If *alias* is the null string (""), the alias field in the user record is not compared.

*u* A reference to a user structure. Upon return, this will be updated with information about the found user record.



|                     |   |
|---------------------|---|
| <b>Return Value</b> | TRUE if the user record was found; FALSE otherwise.   |
| <b>Description</b>  | <p>The <b>userfindopen</b> function is used to initiate a search of the user file. Specifying a name for either <i>name</i> and/or <i>alias</i> instructs Maximus to look for that specific name or alias within the user file.</p> <p>If both <i>name</i> and <i>alias</i> are the null string (""), Maximus will return the first user in the user file.</p> <p>After finding the first record, the <b>userfindnext</b> and <b>userfindprev</b> functions can be used to find the preceding and following user records, thereby allowing the entire user file to be searched.</p> |

---

### userfindprev

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int userfindprev(ref struct _usr: u);</code>  |
| <b>Arguments</b>    | <i>u</i> A reference to a user structure. This should be a structure returned by a previous call to <b>userfindopen</b> , <b>userfindnext</b> or <b>userfindprev</b> . Upon exit, it is updated with a copy of the new user record. |
| <b>Return Value</b> | TRUE if the preceding user record was found; FALSE otherwise.   |
| <b>Description</b>  | The <b>userfindprev</b> function finds the previous record in the user file and returns it in the <i>u</i> structure.   |

---

### userfindseek

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int userfindseek(long: rec, ref struct _usr: u);</code>   |
| <b>Arguments</b>    | <p><i>rec</i>            The record number to be read from the user file</p> <p><i>u</i>              A reference to a user record. Upon return, this structure is updated with a copy of the found user record.</p>  |
| <b>Return Value</b> | TRUE if the user record was successfully retrieved; FALSE otherwise.  |
| <b>Description</b>  | <p>The <b>userfindseek</b> function seeks directly to the specified user record. <b>userfindopen</b> need not be called prior to using <b>userfindseek</b>.</p> <p>To find the following or preceding user records to the record returned by this function, the <b>userfindopen</b> function must be used (with the <i>u.name</i> and <i>u.alias</i> fields set</p> |

appropriately) to find this record, and then **userfindnext** or **userfindprev** can be used to locate subsequent records.

---

**userremove**

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | <code>int userremove(ref struct _usr: u);</code>   |
| <b>Arguments</b>    | <i>u</i> A reference to a user record. This user record is to be removed from the user file. |
| <b>Return Value</b> | TRUE if the record was successfully removed; FALSE otherwise.                                |
| <b>Description</b>  | This function deletes the user contained in the user record specified by <i>u</i> .          |

---

**userupdate**

---

|                     |   |
|---------------------|---|
| <b>Prototype</b>    | <code>int userupdate(ref struct _usr: u, string: origname, string: origalias);</code>   |
| <b>Arguments</b>    | <i>u</i> A reference to a user record. The user specified by the <i>origname</i> and <i>origalias</i> fields will have its user record overwritten by the record specified by <i>u</i> .<br><br><i>origname</i> The original name of the record to be updated.<br><br><i>origalias</i> The original alias of the record to be updated.  |
| <b>Return Value</b> | TRUE if the record was successfully updated; FALSE otherwise.   |
| <b>Description</b>  | The <b>userupdate</b> function is used to update an existing record within the user file. The <i>origname</i> and <i>origalias</i> strings are used as keys for locating the original user record. If the user's name or alias has been changed in the update, the <i>origname</i> and <i>origalias</i> fields must reflect the user's original name and alias, as it was originally read from the user file. |

---

**vidsync**

---

|                     |                              |
|---------------------|------------------------------|
| <b>Prototype</b>    | <code>void vidsync();</code> |
| <b>Arguments</b>    | None                         |
| <b>Return Value</b> | None                         |

**Description**

The **vidsync** function synchronizes screen output with the local video buffer.

**vidsync** is only needed when the *id.instant\_video* variable is set to 0. If *id.instant\_video* is set to 1, Maximus will handle synchronization automatically.

The automatic video synchronization can be disabled since updating the screen can be a slow process. It is often more efficient to disable screen updates, perform a number of functions which modify the screen buffer, and to then call **vidsync** when all of the screen updates are complete.

Note that many of the internal MEX functions that require input (such as **input\_str** and related functions) will automatically update the screen regardless of the *id.instant\_video* setting.

---

**write**


---

**Prototype**

```
int write(int: fd, ref string: s, int: len);
```

**Arguments**

*fd*            A file descriptor, as returned by the **open** function.

*s*             A reference to a string. This string should contain the bytes to be written to the file.

*len*           The number of bytes to be written to the file.

**Return Value**

If the return value is the same as *len*, the write was completely successful.

If the return value is less than *len*, only a portion of the requested number of bytes could be written. The disk is probably full, or some other form of disk error occurred.

**Description**

The **write** function writes a block of bytes to the specified file handle. This function writes blocks of bytes at a time with no consideration for “lines” in the destination file. To write a file a line at a time, see the **writeln** function.

---

**writeln**


---

**Prototype**

```
int writeln(int: fd, string: s);
```

**Arguments**

*fd*            A file descriptor, as returned by the *open* function.

*s*             The string to be written to the file.

|                     |  |
|---------------------|--|
| <b>Return Value</b> | <p>If the return value is equal to <i>strlen(s)</i>, the entire line was written to the file.</p> <p>If the return value is less than <i>strlen(s)</i>, only part of the string could be written. The disk is probably full, or else some other disk error occurred.</p> |
| <b>Description</b>  | The <b>writeln</b> function writes an entire line to the specified file handle. Maximus will automatically append a newline to the end of the line.  |

---

## xfertime

---

|                     |  |
|---------------------|--|
| <b>Prototype</b>    | long xfertime(int: protocol, long: bytes);   |
| <b>Arguments</b>    | <p><i>protocol</i> An index specifying the protocol to use for the purposes of time calculation. This can be <i>usr.def_proto</i> or one of the <i>PROTOCOL_*</i> constants from <i>max.mh</i>.</p> <p><i>bytes</i> The size of the file whose transfer time is to be estimated.</p> |
| <b>Return Value</b> | The estimated number of seconds that will be required to transfer the file.  |
| <b>Description</b>  | The <b>xfertime</b> function is used to estimate the period of time that will be required to download a file of a specific size. The <i>protocol</i> parameter is used to calculate variations in the transfer time based on the efficiency of the various protocols.                |

# 16. MEX Language Reference

## 16.1. Operator Precedence

The following operator precedence is used in MEX expressions. Table 16.1 lists the operators in order from low to high precedence:

Table 16.1 MEX Operator Precedence

| Operator                           | Associativity |
|------------------------------------|---------------|
| <code>:=</code>                    | right to left |
| <code>and or</code>                | left to right |
| <code>= &lt;&gt;</code>            | left to right |
| <code>&lt;= &lt; &gt;= &gt;</code> | left to right |
| <code>shl shr</code>               | left to right |
| <code>&amp;  </code>               | left to right |
| <code>+ -</code>                   | left to right |
| <code>* / %</code>                 | left to right |
| <code>[ ] ( ) .</code>             | left to right |

## 16.2. Language Grammar

This section contains an Extended Backus-Naur Form definition of the MEX grammar:

|                       |           |  |
|-----------------------|-----------|--|
| <i>program</i>        | $\bar{U}$ | <i>top_list</i>  |
| <i>top_list</i>       | $\bar{U}$ | <i>e</i> / <i>top_list func_or_decl</i>                |
| <i>func_or_decl</i>   | $\bar{U}$ | <i>function</i> / <i>declaration</i>                   |
| <i>function</i>       | $\bar{U}$ | <i>typedefn id ( arg_list ) trailing_part</i>          |
| <i>trailing_part</i>  | $\bar{U}$ | <i>function_block</i> / <i>;</i>                       |
| <i>function_block</i> | $\bar{U}$ | <i>{ declarator_list statement_list }</i>              |
| <i>arg_list</i>       | $\bar{U}$ | <i>e</i> / <i>...   argument , arg_list   argument</i> |
| <i>argument</i>       | $\bar{U}$ | <i>opt_ref typename id</i>                             |

|                        |  |  |
|------------------------|--|--|
| <i>opt_ref</i>         | $\dot{U}$  | <b>e</b> / <b>ref</b>  |
| <i>block</i>           | $\dot{U}$  | { <i>declarator_list</i> <i>statement_list</i> }   |
| <i>declarator_list</i> | $\dot{U}$  | <b>e</b> / <i>declarator_list</i> <i>declaration</i>   |
| <i>declaration</i>     | $\dot{U}$  | <i>typename</i> <i>id_list</i> ; / <b>struct</b> <i>id</i> { <i>declarator_list</i> } ;  |
| <i>typename</i>        | $\dot{U}$  | <i>typedefn</i> :  |
| <i>typedefn</i>        | $\dot{U}$<br>$\dot{U}$<br>$\dot{U}$              | <b>char</b>   <b>int</b> / <b>long</b> / <b>signed char</b>   <b>signed int</b>   <b>signed long</b>   <b>unsigned char</b><br><b>unsigned int</b>   <b>unsigned long</b>   <b>void</b>   <b>string</b>   <b>array</b> [ <i>range</i> ] <b>of</b> <i>typedefn</i><br><b>struct</b> <i>id</i>   |
| <i>range</i>           | $\dot{U}$  | < <b>constant_int</b> > .. < <b>constant_int</b> >   < <b>constant_int</b> > ..  |
| <i>id_list</i>         | $\dot{U}$  | <i>id_list</i> , <i>id</i>   <i>id</i>   |
| <i>statement_list</i>  | $\dot{U}$  | <b>e</b> / <i>statement_list</i> <i>statement</i>  |
| <i>opt_statement</i>   | $\dot{U}$  | <b>e</b> / <i>statement</i>  |
| <i>statement</i>       | $\dot{U}$<br>$\dot{U}$<br>$\dot{U}$<br>$\dot{U}$ | ; / <i>block</i> / <i>expr</i> ;   <b>if</b> <i>paren_expr</i> <i>statement</i> <i>else_part</i> / <b>goto</b> <i>id</i> ;<br><i>id</i> : <i>statement</i> / <b>while</b> <i>paren_expr</i> <i>statement</i> /<br><b>do</b> <i>statement</i> <b>while</b> <i>paren_expr</i> ;<br><b>for</b> ( <i>expr</i> ; <i>opt_expr</i> ; <i>opt_expr</i> ) <i>statement</i> / <b>return</b> <i>opt_expr</i> ;   |
| <i>else_part</i>       | $\dot{U}$  | <b>e</b> / <b>else</b> <i>statement</i>  |
| <i>function_call</i>   | $\dot{U}$  | <i>id</i> ( <i>expr_list</i> )   |
| <i>expr_list</i>       | $\dot{U}$  | <b>e</b> / <i>expr</i> / <i>expr</i> , <i>expr_list</i>  |
| <i>primary</i>         | $\dot{U}$<br>$\dot{U}$                           | <i>paren_expr</i> / ( <i>typedefn</i> ) <i>primary</i> / <b>sizeof</b> ( <i>typedefn</i> ) \ <i>function_call</i><br><i>literal</i> / <i>ident</i>   |
| <i>opt_expr</i>        | $\dot{U}$  | <b>e</b> / <i>expr</i>   |
| <i>paren_expr</i>      | $\dot{U}$  | ( <i>expr</i> )  |
| <i>expr</i>            | $\dot{U}$<br>$\dot{U}$<br>$\dot{U}$<br>$\dot{U}$ | <i>expr</i> * <i>expr</i> / <i>expr</i> / <i>expr</i> / <i>expr</i> % <i>expr</i> / <i>expr</i> + <i>expr</i> / <i>expr</i> - <i>expr</i><br><i>expr</i> <= <i>expr</i> / <i>expr</i> < <i>expr</i> / <i>expr</i> <b>shr</b> <i>expr</i> / <i>expr</i> <b>shl</b> <i>expr</i> / <i>expr</i> & <i>expr</i><br><i>expr</i>   <i>expr</i> / <i>expr</i> <b>and</b> <i>expr</i> / <i>expr</i> <b>or</b> <i>expr</i> / <i>expr</i> = <i>expr</i> / <i>expr</i> <> <i>expr</i><br><i>expr</i> >= <i>expr</i> / <i>expr</i> > <i>expr</i> / - <i>primary</i> / <i>primary</i> / <i>ident</i> := <i>expr</i> |
| <i>literal</i>         | $\dot{U}$  | < <b>constant_char</b> >   < <b>constant_int</b> >   < <b>constant_long</b> >   <i>const_string</i>  |
| <i>const_string</i>    | $\dot{U}$  | < <b>constant_string</b> >   <i>const_string</i> < <b>constant_string</b> >  |
| <i>ident</i>           | $\dot{U}$  | < <b>identifier</b> >   <i>ident</i> [ <i>expr</i> ]   <i>ident</i> . <i>id</i>  |
| <i>id</i>              | $\dot{U}$  | < <b>identifier</b> >  |

---

# 17. MECCA Language Reference

This chapter is a guide to the MECCA language. For information on the MECCA compiler itself, please see section 8.8.

## 17.1. Usage Guide

The MECCA language gives the SysOp a large amount of flexibility when designing display screens. MECCA can be used to embed personalized information about the user in text screens, change the text color, run external programs, collect answers to a questionnaire, and perform a variety of other functions.

The input file for the MECCA compiler normally has a **.mec** extension and consists of plain ASCII text. The file can also contain special MECCA tokens which are parsed and translated by the MECCA compiler.

In MECCA, a token is delimited by a set of square brackets (“[” and “]”). Anything outside of square brackets is treated as text and is displayed to the user as-is. Different types of tokens can be inserted in a MECCA file to achieve different purposes. For example, the following line of text:

```
This is your [usercall] call to the system.
```

might be displayed by Maximus as follows, after being compiled with MECCA:

```
This is your 14th call to the system.
```

Other tokens can be used to display the user's name, show information about the current node, and perform conditional actions.

The MECCA compiler only processes tokens that are contained inside of square brackets. (To include a left square bracket in the output, simply insert two left brackets instead of one. Only the left square bracket needs to be doubled.)

For example, to display the following line to a user:

```
Want to check for your mail [Y,n]?
```

Enter this inside a **.mec** file:

```
Want to check for your mail [[Y,n]?
```

When using MECCA tokens, also keep these points in mind:

- Tokens are not case-sensitive. That means that the following tokens are equivalent in all respects:

```
[user]
[USER]
[UsEr]
```

- Spaces are ignored. Inside MECCA tokens, any spaces, tabs or newlines will have no effect. This means that the following tokens are equivalent in all respects:

```
[ user ]
[  user]
[user  ]
```

- More than one token can be inserted inside a set of square brackets, as long as tokens are separated from each other using spaces. For example, this line:

```
[lightblue][blink][user]
```

can also be written as follows:

```
[lightblue blink user]
```

MECCA also allows you to place ASCII codes into the compiled **.bbs** file. To insert a specific ASCII code, simply enclose the ASCII code number inside a pair of square brackets. For example, the token `[123]` will be compiled to ASCII code 123 in the output file.

For conditional testing and flow control, MECCA also allows you to define your own *labels*. When used with the `[goto]` token, labels allow your MECCA file to conditionally display parts of the file, based on user input or other various other conditions.

A label definition looks similar to an ordinary token, except that the label name is preceded by a slash (“/”). The label name must start with a letter, and it must contain only letters, numbers, and underscores. The label name must also be unique, and it cannot use the same name as any existing MECCA token.

A sample label definition looks like this:

```
[/mylabel]
```



When referring to a label as part of a *[goto]* token, you must omit the preceding slash. (The “/” instructs MECCA to mark the file location where the label is placed and use that point as the target for future *[goto]* tokens.

*Example #1.* This MECCA file displays a prompt to the user and requests a *yes* or *no* response. If the user answers “Y,” Maximus displays **c:\max\misc\games.bbs** and asks the question again. Otherwise, Maximus quits the current file. (These MECCA tokens are all described in the following sections, so concentrate on the *[goto]* and the label definitions for now.)

```
[/askgames]Want to play more games? [[Y,n]? [menu]YN
[choice]Y[link]C:\Max\Misc\Games
[choice]Y[goto askgames]
[choice]N[quit]
```

*Example #2.* This code demonstrates a forward-referenced label. The label is actually defined after the *[goto]* token uses it:

```
Want a "Zippy the Pinhead" quote? [[y,n]? [menu]YN
[choice]Y[goto zippy]
[choice]N[quit]
[/zippy]
Okay, here's the quote!
[quote quit]
```

For more examples, look at some of the \*.mec files in the \max\hlp and \max\misc directories.

## 17.2. Color Token Listing

MECCA allows you to use up to 128 different color combinations for displaying text. To display text in a different color, simply enclose the name of the color in square brackets. For example, *[yellow]* displays the following text in yellow. (A complete list of permissible colors is given later in this section.)

To display text on a colored background, use a token of the form “**fore on back**”; **foreground** is the name of the foreground color and **background** is the name of the background color.

For example, to display light green text on a blue background, enter this:

```
[lightgreen on blue]
```

|                       |
|-----------------------|
| Title: Af<br>Creator: |
|-----------------------|

Only the first eight normal colors (see Table 17.1 below) can be used for the background color. The colors beginning with “light,” “dark,” “white” and “yellow”) can only be used as foreground colors.

MECCA also supports blinking text. To make text blink, simply insert the `[blink]` token after the color token. Text that follows this token will blink. (A color token always resets a previous `[blink]` token, so if you place the `[blink]` token immediately before a color token, the `[blink]` token will have no effect.)

For example, the following line displays blinking green text:

```
[green blink>Hello, world!
```

However, this line displays non-blinking text:

```
[blink green>Hello, world!
```

If the user has disabled ANSI or AVATAR graphics support, Maximus will automatically strip out the color and cursor-movement codes before transmitting the screen to the user. Consequently, the same colorized screen can be shown to users with either TTY or ANSI/AVATAR terminals.

Table 17.1 lists the colors supported by MECCA:

**Table 17.1 MECCA Colors**

| Foreground and Background | Foreground only |
|---------------------------|-----------------|
| [black]                   | [darkgray]      |
| [blue]                    | [lightblue]     |
| [green]                   | [lightgreen]    |
| [cyan]                    | [lightcyan]     |
| [red]                     | [lightred]      |
| [magenta]                 | [lightmagenta]  |
| [brown]                   | [yellow]        |
| [gray]                    | [white]         |

Other tokens relating to colors are:

`[BG <c>]`

This token is a MECCA directive that modifies the current background color without changing the foreground color. The new background color is set to `<c>`.

For example, this code:

```
[red on blue>Hello, [BG green]user
```

displays the text “Hello,” in *red on blue*, while it displays “user” in *red on green*.

*[blink]* - ^v^b

Any text that follows this token will blink. The blinking attribute is reset when Maximus encounters a color token.

*[bright]*

This token is a MECCA directive that sets the intensity bit of the current color. If the current foreground color is on the left hand side of Table 17.1, MECCA sets the new color to the equivalent color on the right hand side of the table.

For example, the following code:

```
[red]Is it not a [lightred]BEAUTIFUL DAY?
```

can be replaced with this simpler form:

```
[red]Is it not a [bright]BEAUTIFUL DAY?
```

*[dim]*

This token is a MECCA directive that disables the intensity bit for the current color. Similar to the *[bright]* token, it translates colors from the right hand side of the color table to the equivalent color on the left hand side of the table.

For example, this code:

```
[lightgreen]H[dim]e[bright]l[dim]l[bright]o[dim]!
```

displays the word “Hello!”, with each character alternating between normal green and light green.

*[FG <c>]*

This token is a MECCA directive that modifies the current foreground color without changing the background color. The new foreground color is set to **<c>**.

For example:

```
[lightred on blue]Hi, [FG yellow]Mr. Smith...
```

The above line displays the text “Hi,” in *lightred on blue*, and it displays the phrase “Mr. Smith” in *yellow on blue*.

*[load]*

This token is a MECCA directive which restores the color that was previously saved using the *[save]* token.

These two tokens are useful when creating screens with backgrounds, since repeatedly typing two different colors can get verbose. (To alternate between two colors, you can save the first color by using *[save]*, and after inserting the command to change to another color, you can restore the first color by inserting a *[load]* token.)

For example:

```
[yellow on blue save]This is yellow on blue.[cleol]
[lightred on green]This is lightred on green.[cleol]
[load]This text is also yellow on blue.[cleol]
```

*[on]*

This is a MECCA directive which tells MECCA to interpret the next token as a background color. See the introduction at the beginning of this section for more information.

*[save]*

This keyword is a MECCA directive that tells MECCA to save the current color and store it for retrieval by the *[load]* token.

*[steady]*

This keyword is a MECCA directive that disables a previous *[blink]* token.

For example:

```
[yellow]This does not blink. [blink]This does.
[steady]However, this text is non-blinking.
```

### 17.3. Cursor Control and Video Tokens

This section describes video and terminal control commands that can be used to move the cursor and manipulate the caller's screen.

*[bell]* - ^g

This causes a beep (ASCII 07) to be generated on the user's terminal.

*[bs]* - ^h

This causes a backspace (ASCII 08) to be generated on the user's terminal. This token moves the cursor left by one column.

*[cleol]* - ^v^g

This instructs Maximus to send a *clear to end of line* command. If the current background color is non-black, the rest of the line is set to that color.

*[cleos]* - ^v^o

This instructs Maximus to clear out part of the screen, from the current cursor position to the end of the screen. The cleared screen is set to the current color, and the cursor position is not changed. This is not a true AVATAR sequence, but Maximus will automatically generate the appropriate commands to clear the screen.

*[cls]* - ^l

This clears the user's screen and sets the current color to cyan.

*[cr]* - ^m

This sends a carriage return to the user.

*[down]* - ^v^d

This tells Maximus to move the cursor by down one row.

*[left]* - ^v^e

This moves the cursor one column to the left.

*[lf]* - ^j

This sends a linefeed to the user.

*[locate <r> <c>]* - ^v^h<r><c>

This command moves the cursor to the <r>th row and the <c>th column of the screen. (The top left corner of the screen is row 1, column 1.)

*[tab]* - ^i

This command sends a tab character to the user.

*[right]* -  $\wedge v \wedge f$

This moves the cursor one column to the right.

*[sysopbell]* -  $\wedge w \wedge g$

This token sounds a beep on the local console. The beep is not transmitted to the on-line user.

*[up]* -  $\wedge v \wedge c$

This moves the cursor up by one row.

## 17.4. Informational Tokens

Maximus includes a large number of tokens that display selected information about the user and about the system in general. Maximus supports the following informational tokens:

*[alist\_file]* -  $\wedge r l F$

Display the file area menu. If a **Uses FileAreas** file is defined in the system control file, this token will display it. Otherwise, Maximus will automatically build an area list and display it to the user.

*[alist\_msg]* -  $\wedge r l M$

Display the message area menu. If a **Uses MsgAreas** file is defined, this token will display it. Otherwise, Maximus will automatically build an area list and display it to the user.

*[city]* -  $\wedge f \wedge c$

Display the user's city.

*[date]* -  $\wedge f \wedge d$

Display the current date in the “dd mmm yy” format.

*[dl]* -  $\wedge f \wedge x$

Display the user's total number of kilobytes downloaded, including today's downloads.

*[expiry\_date]* - ^wyD

Display the user's current expiration date, or "None" if the user has no expiration date.

*[expiry\_time]* - ^wyT

This displays the time left in the current user's subscription. If the user has time remaining, this token displays "x minutes," where *x* is the number of minutes remaining in the user's subscription. If the user has no time subscription, this token displays "None."

*[file\_carea]* - ^w^fA

Display the name of the current file area.

*[file\_cname]* - ^w^fN

Display the description for the current file area.

*[file\_darea]* - ^w^fD

Display the name of the current file area division.

*[file\_sarea]* - ^w^fd

Display the name of the current file area (without the division prefix).

*[fname]*

*[first]* - ^f^f

Display the user's first name.

*[lastcall]* - ^w^a

Display the date of the user's last call.

*[lastuser]* - ^w^k

Display the name of the last user to call the current node. This information is read from the **bbstat##.bbs** file from the Maximus system directory.

*[length]* - ^f^l

Display the duration of this user's call, in minutes.

## 344 17. MECCA Language Reference

*[minutes]* -  $^f k$

Display the number of minutes for which the user has been on-line during the last 24 hours.

*[msg\_carea]* -  $^w mA$

Display the name of the current message area.

*[msg\_cmsg]* -  $^w mL$

Display the current message number.

*[msg\_cname]* -  $^w mN$

Display the description for the current message area.

*[msg\_darea]* -  $^w mD$

Display the name of the current message area division.

*[msg\_hmsg]* -  $^w mH$

Display the highest message number in the current area.

*[msg\_nummsg]* -  $^w m\#$

Display the number of messages in the current area.

*[msg\_sarea]* -  $^w md$

Display the current message area (without the division prefix).

*[netbalance]* -  $^w nB$

Display the current user's matrix balance (credit minus debit) in cents.

*[netcredit]* -  $^w nC$

Display the current user's matrix credit in cents.

*[netdebit]* -  $^w nD$

Display the current user's matrix debit in cents.



*[netdl]* - ^f^r

This displays the user's net downloads for today (today's downloads minus today's uploads).

*[node\_num]* - ^wjN

Display the current node number.

*[phone]* - ^wP

Display the user's phone number.

*[ratio]* - ^f^y

Display the current user's download ratio, in the format of **uploads:downloads**.

*[realname]* - ^wR

Display the user's real name (if applicable).

*[remain]* - ^f^o

Display the number of minutes that the user has left for the current call.

*[response]* - ^w^e

Display the last line entered by the user for a *[readln]* response. This token even works across files — if one file contain a *[readln]* token, the *[response]* token can be inserted in a separate file to display the result. See also *[ifentered]*.

*[syscall]* - ^f^q

Display the total number of calls that the current node has received (as an ordinal number).

*[sys\_name]* - ^r^c

Display the system name.

*[sysop\_name]* - ^r^d

Display the SysOp's full name.

*[time]* -  $\backslash^f t$

Display the current time in the format “hh:mm:ss”.

*[timeoff]* -  $\backslash^f p$

Displays the time by which the user must be off the system. This string includes a newline at the end; unless you want to have a blank line displayed in your output file, you should not press *<enter>* immediately after entering this token in the MECCA source file.

*[ul]* -  $\backslash^f w$

Display the count of kilobytes uploaded, including today's statistics.

*[user]* -  $\backslash^f b$

Display the user's full name.

*[usercall]* -  $\backslash^f e$

Display the number of times the current user has called your system (as an ordinal number).

## 17.5. Questionnaire Token Listing

The tokens in this section can be used to design on-line questionnaires and to log information to a file. All of the questionnaire tokens are described later in this section, but most questionnaires will follow the general format given below:

One of the first tokens in a questionnaire file should be the *[open]* token. This token opens the specified filename for output, and this is where Maximus logs all of the questionnaire answers. (This file is human-readable, so you can view the file with a normal text editor.)

A *[post]* token normally follows the *[open]* token. The *[post]* writes the user's name, city, and the current date/time to the questionnaire file. (For an anonymous questionnaire, this step can be omitted.)

You can then insert the main body of the questionnaire. The *[readln]* token is used to request input from the user. All of the input lines are written to the questionnaire file that was opened using *[open]*.

The *[store]* can also be used to store the response to a menu displayed by *[menu]*.

Finally, you can also use the *[write]* to write lines directly to the questionnaire file. These lines can include external program translation characters, as described in section 6.5.

You can place as many questions in a questionnaire as desired. These questionnaire tokens can be placed in any **.mec** file.

The following tokens are related to designing questionnaires:

*[ansopt]* - ^f^v

After this token is encountered, Maximus will not require an answer for *[menu]* and *[readln]* tokens. The user can simply press <enter> to skip the prompt.

*[ansreq]* - ^f^u

After this token is encountered, Maximus will require an answer for all *[menu]* and *[readln]* tokens.

*[choice]<c>* - ^oU<c>

Display the rest of the current line only if the response to the last *[menu]* choice is equal to the character <c>.

*[leave\_comment]* - ^wK

Place the user in the message editor and allow the user to write a message to the SysOp. The message is saved in the area defined by **Comment Area** in the system control file.

If the message is aborted by the user, or if the message saved by the user is blank, Maximus will skip the rest of the line containing the *[leave\_comment]* token.

This construct allow you to determine if the user entered a message and react accordingly:

```
Please leave a comment to the SysOp, [fname].
[enter]
[/Do_Comment leave_comment goto Successful]
```

```
You did not leave a real message! Try again...
[enter goto Do_Comment]
```

```
[/Successful]Thanks for leaving a comment, [fname].
```

*[menu]*<k> - ^oR<k>

This token prompts the user to press a key. The value entered by the user can be later manipulated using the *[choice]* and *[store]* tokens.

<k> is a list of valid keys that the user can use to respond to the menu. If the *[ansopt]* token is in effect, the user can also press <enter> to skip the question.

If the user enters a key which is not in <k>, Maximus displays an error message and prompts the user to try again. The characters in <k> can be any character between ASCII 33 and ASCII 126, including letters, numbers and punctuation marks.

*[open]*<f> - ^oO<f>

This command instructs Maximus to open a questionnaire answer file called <f>. See also *[post]*, *[store]* and *[readln]*. Maximus honors external program translation characters in this filename.

*[post]* - ^oP

Write the user's name, city, and the current time/date to the questionnaire answer file.

*[readln]*<d> - ^oN<d>

Retrieve a line of input from the user, and then write it to the questionnaire answer file, placing the optional one-word description <d> beside the user's answer.

By default, the *[readln]* token allows stacked commands. For example, if a user entered input that was not used by a prior prompt, any remaining input in the key input buffer will be read by the *[readln]* command. To disable this functionality, include a *[clear\_stacked]* token just before the *[readln]* token.

*[sopen]*<f> - ^oo<f>

Open a questionnaire answer file called <f>. This token is identical to the *[open]* token.

*[store]*<d> - ^oM<d>

Writes the user's response to the last *[menu]* token into the questionnaire answer file, placing the optional one-word description <d> beside the user's answer.

*[write]<l> - ^wW<l>*

Write the line <l> directly to the questionnaire answer file, interpreting any external program translation characters contained within. Please see section 6.5 for more information.

## 17.6. Privilege Level Controls

These tokens allow certain parts of a MECCA file to be displayed or skipped, depending on the user's current privilege level.

*[?below], [?equal], [?file], [?line], [?xclude], [above], [ae], [be], [below], [eq], [equal], [ge], [gt], [le], [lt], [ne], [notequal], [unequal]*

These keywords are obsolete. These keywords are only supported for compatibility with previous versions of Maximus.

*[acs <acs-string>]  
[access <acs-string>] - ^pa<acs-string><space>*

Display the rest of the line only if the user's access level passes the ACS given by <acs-string>. Any type of ACS test can be included here.

*[acsfile <acs-string>]  
[accessfile <acs-string>] - ^pf<acs-string><space>*

Display the rest of the file only if the user's access level passes the ACS given by <acs-string>. Any type of ACS test can be included here.

*[priv\_abbrev] - ^vpa*

Display the user's class level abbreviation. (For example, this could display "SysOp" or "AsstSysOp," depending on the definition in the access control file.)

*[priv\_desc] - ^vpd*

Display the user's class level description, as defined in the access control file.

*[priv\_down] - ^wpD*

Lower the privilege level of the current user to that of the next lower user class.

*[priv\_level] - ^vpI*

Display the user's numeric privilege level.

*[priv\_up] - ^wpU*

Raise the privilege level of the current user to that of the next higher user class.

*[setpriv <priv>] - ^ws?*

Adjusts the current user's privilege level to a certain value. **<priv>** should be the single-character key value associated with a user class defined in the access control file.

## 17.7. Lock and Key Control

*[ifkey]<keys> - ^wkI*

If the specified keys are set, the rest of the line is displayed. Any number of keys can be specified, but they must be separated from the rest of the line by a space. For example:

*[ifkey]123a You have keys 1, 2, 3 and A.*

*[notkey]<keys> - ^wkN*

This token is similar to *[ifkey]*, except that the line is displayed only if the specified keys are not set.

For example:

*[notkey]8b You have neither key 8 nor key b.*

*[keyon]<keys> - ^wkO*

This command gives the specified keys to the user. **<keys>** must be separated from the rest of the line with a space.

*[keyon]6abc User, you now have keys A, B, C and 6.*

*[keyoff]<keys> - ^wkF*

This command takes the specified keys away from the user. **<keys>** must be separated from the rest of the line with a space.

This command turns *off* the specified keys.

[keyoff]fgh User, keys F, G and H have been removed.

## 17.8. Conditional and Flow Control Tokens

These tokens allow you to conditionally display different parts of a file, depending on various user attributes and system settings.

*[b1200] - ^w^b1*

Display the rest of the line only if the user is connected at 1200 bps or above.

Using this construct, you can use the *[b1200]* token to display a line only to users who are calling at a speed slower than 1200 bps:

```
[b1200 goto FastUser]
You are a 300 baud user!
[goto Done]

[/FastUser]
You are a 1200 bps or above user!
[/Done]
```

*[b2400] - ^w^b2*

Display the rest of the line only if the user is connected at 2400 bps or above.

*[b9600] - ^w^b9*

Display the rest of the line only if the user is connected at 9600 bps or above.

*[col80] - ^w8*

Display the rest of the line only if the user's screen is at least 79 columns wide.

*[color] - ^oE*

*[colour] - Canadian spelling of above*

Display the following text (up to the next *[endcolor]* token), only if the user has ANSI or AVATAR graphics. See also *[nocolor]*.

*[endcolor] - ^oe*

*[endcolour] - Canadian spelling of above*

This token marks the end of a “color-only” display sequence. See also *[color]*.

*[endrip]* - ^oI

This token marks the end of a “RIP<sub>scrip</sub>-only” display. See also *[rip]*.

*[expert]* - ^wHE

Display the rest of the line only if the user’s current help level is *expert*.

*[exit]* - ^wE

Exit all display files, including those which were displayed using a *[link]* token. See also *[quit]*.

*[filenew]* <f> - ^wf

Display the rest of the current line if the file <f> is newer than the user’s last log-on date. The file <f> must be separated from the rest of the line by a space. This means that a construct such as this can be used to display a file only if the user has not seen it before:

```
[filenew]c:\max\misc\bulletin.bbs [display]c:\max\misc\bulletin.bbs
```

*[goto <l>]* - ^oV

Jump to the label <l> in the current file.

*[hotkeys]* - ^rh

Display the rest of the current line only to those users who have hotkeys enabled.

*[ifentered]* <s> - ^we<s>

This token compares <s> to the response given by the user to the last *[readln]* token. If the two strings are equal, the rest of the line is displayed. <s> is separated from the rest of the line by a single space.

For example, given the following code:

```
What kind of yogurt do you like best? [readln]
[ifentered]peach You are a real peach!
[ifentered]lemon You are what you eat!
```

If the user enters “peach” at the prompt, Maximus will display “You are a real peach!” If the user enters “lemon,” Maximus will display “You are what you eat!” If the user entered neither string, Maximus would display nothing.



*[ifexist]<filename> - ^wi<filename>*

The rest of the line is displayed only if the file **<filename>** exists. The filename must be separated from the rest of the line by a space.

*[iffse] - ^w^mE*

Display the rest of the line only if the user has enabled the full-screen editor.

*[iffsr] - ^w^mR*

Display the rest of the line only if the user has the full-screen reader enabled.

*[iflang]<l> - ^wB<l>*

Display the rest of the line if the user's language is set to language number **<l>**. The language number is 0-based, so a "0" means the first language listed in the language control file, "1" means the second language, and so on.

*[iftask]<tasknum> - ^wb<tasknum>*

Display the rest of the line if the current node is **<tasknum>** (in decimal). The task number is separated from the rest of the line by a space.

*[iftime <op> <hh>:<mm>]*

Display the rest of the line only if the current time of day matches a specified condition.

**<op>** specifies the type of comparison operation to perform. The supported operators are listed below in Table 17.2:

**Table 17.2 [iftime] Operations**

| Operator       | Description  |
|----------------|--|
| EQ or EQUAL    | Display the rest of the line only if the current time equals hh:mm.                        |
| NE or NOTEQUAL | Display the rest of the line only if the current time is not equal to hh:mm.               |
| LT or BELOW    | Display the rest of the line only if it is currently earlier than (and not exactly) hh:mm. |
| GT or ABOVE    | Display the rest of the line only if it is currently later than (and not exactly) hh:mm.   |
| GE or AE       | Display the rest of the line only if it is currently later than (or equal to) hh:mm.       |
| LE or BE       | Display the rest of the line only if it is earlier later than (or equal to) hh:mm.         |

**<hh>** and **<mm>** tell Maximus which hour and minute to compare with, specified in 24-hour format.

For example:

```
[iftime GE 20:00]It is after 8 pm!
[iftime GE 20:00 iftime LE 21:00]Between 8 and 9 pm.
[iftime LT 20:00 iftime NE 12:00]Before 8 & not 12.
```

*[incity]<s> - ^wR*

Display the rest of the line if the string **<s>** can be found in the user's city field. **<s>** should be separated from the rest of the line by a single space.

Example:

```
[incity]Toronto Hi, [first]. You are a Torontonion!
```

*[islocal] - ^wIL*

Display the rest of the line only if the user is logged in at the local console.

*[isremote] - ^wIR*

Display the rest of the line only if the user is logged in from remote.

*[jump] - ^oV*

This token is a synonym for *[goto]*.

*[label <l>]*

This token provides an alternate way to define a label. The sequence "[label **<l>**]" is identical to "[/**<l>**]".

*[maxed] - ^wm*

Display the rest of the line only if the user is currently in the MaxEd editor. This token can be useful when designing a custom menu for the editor.

*[msg\_attr <letter>] - ^w^mB<letter>*

This token displays the rest of the line only if the user has sufficient access rights to create a message with the specified attribute type.

<letter> must correspond to a character in the **msgattr\_keys** variable in the system language file. The default keys for the English language are listed below in Table 17.3:

**Table 17.3 [msg\_attr] Keys**

| Key | Attribute       |
|-----|-----------------|
| P   | Private         |
| C   | Crash           |
| R   | Received        |
| S   | Sent            |
| A   | Attach (file)   |
| F   | Forward         |
| O   | Orphan          |
| K   | Kill/Sent       |
| L   | Local           |
| H   | Hold            |
| X   | Xx2 (undefined) |
| G   | File Request    |
| !   | Receipt Request |
| \$  | Return Receipt  |
| T   | Trail Request   |
| U   | Update Request  |

For example:

```
[                ]Key      Action
[                ]=====
[msg_attr P]P      To toggle the PRIVATE flag
[msg_attr C]C      To toggle crash status
[msg_attr A]A      To attach a file
[msg_attr R]R      To toggle the Received flag
[msg_attr S]S      To toggle the Sent flag
[msg_attr F]F      To toggle the Forward flag
[msg_attr O]O      To toggle the Orphan flag
[msg_attr K]K      To kill the message after packing
[msg_attr H]H      To keep message on hold for pickup
[msg_attr G]G      To request a file
[msg_attr !]!      To request a receipt
[msg_attr $]$      To toggle the receipt flag
[msg_attr T]T      To request a trail
[msg_attr U]U      To place an update request
[                ]<enter> To proceed to the next field
[white enter quit]
```

*[msg\_conf] - ^w^maC*

Display the rest of the line only if the current message area is a conference area.

*[msg\_echo]* - ^w^maE

Display the rest of the line only if the current message area is an EchoMail area.

*[msg\_fileattach]* - ^w^mF

Display the rest of the line only if the user has pending file attaches in the message area. This token will not detect inbound FTS-0001 style attaches.

For example:

*[msg\_fileattach][yellow blink]*You have files attached!

*[msg\_local]* - ^w^maL

Display the rest of the line only if the current message area is a local area.

*[msg\_matrix]* - ^w^maM

Display the rest of the line only if the current message area is a NetMail area.

*[msg\_next]* - ^w^miN

Display the rest of the line only if the current message-reading direction is *forward* (as opposed to *backward*).

*[msg\_nomsgs]* - ^w^mnM

Display the rest of the line only if the current message area contains no messages.

*[msg\_nonew]* - ^w^mnN

Display the rest of the line only if the current message area does not contain any new messages.

*[msg\_noread]* - ^w^mnR

Display the rest of the line only if the user has not read any of the messages in the current message area.

*[msg\_prior]* - ^w^miP

Display the rest of the line only if the current message-reading direction is *backward* (as opposed to *forward*).

*[no\_keypress]* - ^wG

Display the rest of the line only if there are no keystrokes waiting in the input buffer. See also *[nostacked]*. This differs from *[nostacked]* in that *[no\_keypress]* only checks for currently-pending input which has not been processed, whereas *[nostacked]* checks for commands entered at a previous prompt.

*[nocolor]* - ^o^^

*[nocolour]* - Canadian spelling of above

Display the following text (up to the next *[endcolor]* token) only to callers who do not have ANSI or AVATAR graphics selected.

*[norip]* - ^oH

Display the following text (up to the next *[endrip]* token) only to callers who do not have RIP<sub>scrip</sub> graphics selected.

For example:

```
[norip]You are not a RIP caller![endrip]
```

*[nostacked]* - ^wS

Display the rest of the line only if there are no keystrokes waiting in the key stacking buffer. See also *[no\_keypress]*.

*[notontoday]* - ^wQ

Display the current line only if the user has not logged on the system previously in the day. (The user must have been logged on for at least one minute for this token to take effect.)

*[novice]* - ^wHN

Display the current line only if the user's help level is set to *novice*.

*[ofs]*

This token is obsolete.

*[permanent]* - ^wq

Display the current line only if the current user is marked as being *permanent*. The permanent flag indicates that the user should not be deleted by any automatic, third-party user purging utilities.

*[regular]* - ^wHR

Display the rest of the line only if the user has a help level of *regular*.

*[rip]* - ^oG

Display the following text (up until the next *[endrip]* token) only to those users who have *RIPscrip* graphics enabled. Note that *RIPscrip* graphics are never shown on the local console, regardless of the caller's *RIPscrip* setting.

*[rip]*You are a RIP caller!*[endrip]*

*[riphasfile]*<name> [,<size>] - ^wY

Display the rest of the line only if the remote caller:

- ◆ has *RIPscrip* graphics enabled, and
- ◆ has already received the file <name> with a size of <size>.

The “,<size>” part of the argument is optional. If a <size> is provided, Maximus will also check the size of the file on the remote system and ensure that it matches the specified size. If it does not, Maximus will act as if the file did not exist.

The “<name>,<size>” argument must be separated from the rest of the line by a space.

Example #1:

```
[riphasfile]m_menu.rip,872 |1R00000000m_menu.rip[quit]
```

This code queries the remote system for a file called **m\_menu.rip** that is 872 bytes in length. If the file exists, Maximus displays the rest of the line (which contains the *RIPscrip* sequence to display that file).

Example #2:

```
[riphasfile]m_menu.rip !|1R00000000m_menu.rip[goto quitfile]
[link]rip\m_menu
[/quitfile]
```

This code queries the remote system for a file called **m\_menu.rip** (of any size). If the file exists, Maximus displays the rest of the line.

This token can be used in conjunction with the *[ripsend]* token to allow users to *optionally* download all of the *RIPscrip* scenes used on your BBS.

*[tagged]* - ^w^

Display the rest of the line only if the user has one or more files currently tagged.

*[top]* - ^oT

Jump back to the top of the file and start the file display over again.

## 17.9. Multinode Tokens

This section describes MECCA tokens that may be useful for systems with more than one node:

*[apb]* - ^wA

This token sends a message to all users currently on-line, assuming that the IPC features are enabled.

Example #1:

```
[apb][yellow bell]%!User %n just logged on the system%!
```

Example #2:

```
Enter a message to send to all users: [readln]
[apb][yellow bell]%!User %n says "%J"%!
```

*[chat\_avail]* - ^wcA

Display the rest of the line only if the user is available for paging by other users.

*[chat\_notavail]* - ^wcN

Display the rest of the line only if the user is not available for paging by other users.

*[who\_is\_on]* - ^ww

Execute the internal **Who\_Is\_On** menu command.

## 17.10. RIPscrip Graphics

This section describes tokens that are useful when creating RIPscrip graphics displays. See also the *[riphasfile]*, *[rip]*, *[norip]* and *[endrip]* tokens.

*[ripdisplay]* <file> - ^wv

Instruct the remote system to immediately display a RIPscrip icon or scene file.

If the remote system does not have the specified file, *[ripdisplay]* will automatically send it. Please see *[ripsend]* for information on where Maximus looks to find the file to be sent.

*[rippath]* <path> - ^wU

Change the directory where Maximus looks to find RIPscrip scene files and icons. The default is the **RIP Path** specified in the system control file.

For example:

```
[iflang]1 [rippath]C:\Max\Rip.1
```

*[ripsend]* <file> - ^wV

Send one or more icons or RIPscrip scene files to the remote system for storage only. These files must exist in the **RIP Path** directory or the directory specified by the most recent *[rippath]* token. (If the file does not exist in the directory specified by *[rippath]*, Maximus will look for the file in the default **RIP Path** directory.)

If the filename specified for *[ripsend]* or *[ripdisplay]* starts with “@”, the file is assumed to be an ASCII file on the local system that contains a list of files to be sent.

Otherwise, <file> must be a simple filename (with no path) for a file in the current RIPscrip directory, as specified by the **RIP Path** statement in the system control file.

If <file> starts with “!”, Maximus ignores the fact that the file may have already been sent during the current session, and it will query the remote system anyway (and send the file if the user does not have it).

Examples:

```
[ripsend]menu.rip f_icon.icn m_icon.icn
```

This line simply sends the named files.



```
[ripsend]@c:\max\allicons.lst
```

This line sends all files listed in the **c:\max\allicons.lst** file.

```
[ripsend]!menu.rip f_icon.icn m_icon.icn
```

This line sends all files in the named list, regardless of whether or not Maximus has sent them previously in the session. (However, if the remote user has a matching file name and size, Maximus will not send the file.)

```
[textsize <cols> <rows>] - ^wg
```

Set the assumed text window size of the remote system. This token can be used to make “More [Y,n,=]?” prompts appear in the correct position when working with a *RIPscrip* scene that modifies the text window size.

Examples:

```
[textsize 0 0]
```

This resets the window size back to the default from the user file.

```
[textsize 60 0]
```

This sets the window width to 60 and sets the window height to the default from the user file.

```
[textsize 0 32]
```

This sets the window height to 32 rows and sets the window width to the default from the user file.

This MECCA command should be used whenever a called *RIPscrip* scene reduces the size of the remote text window.

## 17.11. Miscellaneous Token Listing

This section contains miscellaneous MECCA tokens that are not specific to one of the other sections:

```
[ckoff] - ^b
```

Disable *<ctrl-c>* and *<ctrl-k>* checking. This prevents the user from using these keys to abort the current file.

*[ckon]* - ^c

Enable `<ctrl-c>` and `<ctrl-k>` checking. This allows the user to press these keys to abort the current file.

*[clear\_stacked]* - ^wO

Clear the user's command stacking buffer to eliminate any previously-stacked commands.

*[comment <c>]*

This token is a comment. Anything enclosed in a comment token is ignored by the MECCA compiler.

*[copy <f>]*

This token is a MECCA directive. When MECCA encounters this token, it copies the file `<f>` directly into the output file, without performing any translations or parsing. Ensure that the filename `<f>` is inside the square brackets.

*[decimal]*

This token is obsolete.

*[delete]<f>* - ^wD<f>

Delete file `<f>` from disk. External program translation characters may be used here.

*[display]<f>* - ^oS<f>

Display the named **.bbs** file. Control is not returned to the current file after `<f>` has displayed. External program translation characters may be used in the filename for `<f>`, but you must use a “+” instead of a “%” to begin the translation sequence. See also *[link]*.

*[dos]<c>* - ^oC<c>

Run the operating system command `<c>`. This command can include arguments and external program translation characters. See also *[xtern\_\*]*.

*[enter]* - ^a

Display “Press ENTER to continue” and wait for the user to press `<enter>`.

*[hangup]* - ^f^n

Hang up and disconnect the current user.

*[hex]*

This token is obsolete.

*[ibmchars]* - ^wd

Display the rest of the line only to those users who have the IBM Characters option enabled.

*[include <f>]*

This token is a MECCA directive. MECCA reads in the file <f> and processes it as if it were part of the current file.

*[key?]*

This token is obsolete.

*[key\_poke]<keys>* - ^wP

Insert keystrokes into the keyboard command-stacking buffer, just as if the user had entered the keys manually. This token can be used to automatically guide a user through several commands at once.

The <keys> sequence can include a “|” character. This character will be translated into an empty response.

For example, the following MECCA sequence skips over the “enter date:” prompt in the new files scan:

```
[key_poke]|
[newfiles]
```

*[language]* - ^oL

Invoke the internal **Chg\_Language** menu option.

*[link]<f>* - ^wL

Display the specified .bbs file. Control is returned to the current display file after <f> has displayed. Up to 8 display files can be nested using the *[link]* token.

*[log]<s> - ^wA<s>*

Add the statement *<s>* to the system log. The first character of *<s>* must be the log “priority level” for the left-hand column of the log. The rest of *<s>* is the log string to insert. External program translation characters can also be inserted in this string.

For example:

```
[log]+User's name is "%n"
```

*[menu\_cmd <s>]<arg> - ^rr*

Invoke the menu command specified by *<s>*.

For example, *[menu\_cmd goodbye]* invokes the “Goodbye” menu option. Please see section 18.8 for a list of valid menu options.

*[menupath]<p> - ^wM<p>*

Sets the current path for \*.**menu** files to *<p>*.

*[mex]<file> - ^wn*

Run the MEX program specified by *<file>*. For example:

```
[mex]d:\path\filename arg1 arg2 arg3
```

*[more] - ^d*

Display a “More [Y,n,=]?” prompt.

*[moreoff] - ^k*

Disable the automatic “More [Y,n,=]?” prompting feature.

*[moreon] - ^e*

Enable the automatic “More [Y,n,=]?” prompting feature.

*[msg\_checkmail] - ^wC*

Invoke the internal mail checker.

*[newfiles] - ^wF*

Invoke a new files scan.

*[onexit]*<f> - ^oF<f>

This sets the *On Exit* filename for the current display file. When the current file has finished displaying, Maximus will display the file indicated by <f>.

*[pause]* - ^f^g

Pause for half a second.

*[quit]* - ^oQ

Quit the current file immediately. If the current file was displayed with a *[link]* token, this command will exit back to the calling display file. See also *[exit]*.

*[quote]* - ^f^a

Display the next quote from the file in the **Uses Quote** definition in the system control file.

*[repeat]*<c>[<n>] - ^y<c><n>

Output a sequence of repeated bytes. Maximus will output the character <c> a total of <n> times. The value for <n> must normally be contained in square brackets.

For example, this text

```
[repeat]=[15]
```

displays the “=” character 15 times.

*[repeatseq <len>]<s>[<n>]* - ^v^y<len><s><n>

Repeat the text string <s> a total of <n> times. The string <s> must have a length of <len>.

For example:

```
[repeatseq 7]Hello! [10]
```

The above line displays the phrase “Hello!” followed by a space (a total of ten times).

*[string]*  
*[subdir]*  
*[unsigned]*

These tokens are obsolete.

*[tag\_read]<filename> - ^wz*

Read a list of tagged files from **<filename>**. The current list of tagged files is destroyed.

*[tag\_write]<filename> - ^wZ*

Write the current list of tagged files to **<filename>**. If no filename is specified, Maximus writes the files to the file specified by the **default\_tag\_save** field in the system language file. External program translation characters may be used in this string.

*[tune]<name> - ^wu<name>*

Play the specified tune from the **tunes.bbs** file on the local console.

For example:

```
[tune]Yell11
```

*[xtern\_dos]<c> - ^wXD<c>*  
*[xtern\_erlwl]<c> - ^wXE<c>*  
*[xtern\_run]<c> - ^wXR<c>*

Run the external program **<c>**. If you wish to pass arguments to the external program, simply add the arguments after the name of the program to execute, separating each argument with a space. Please see section 6 for more information on running external programs with Maximus.

---

# 18. Control File Reference

The system control files, `\max\*.ctl`, are the heart of your BBS. This reference section describes all of the keywords and commands contained within. The keywords are organized by control file and are listed alphabetically.

The control files must be compiled using the SILT compiler before Maximus will recognize any changes that you may make. Please see section 8.12 for more information on the SILT compiler.

## 18.1. SILT Directives

Directives are commands that are interpreted only by the SILT compiler; they simply modify SILT's operation and do not produce any output on their own.

These directives can be used in any configuration file:

*Include* <filespec>

The **Include** keyword instructs SILT to process the file <filespec> as if it were included as part of the current control file. The **Include** keyword cannot be used within a **Section** statement.

*Version14* <filespec>

*Version17* <filespec>

These keywords are obsolete.

## 18.2. System Control File

The system control file, `max.ctl`, is the most important part of any Maximus system. A heavily-commented system control file comes as part of the standard Maximus installation, and in many cases, you can rely on only the comments contained within. However, this section provides a comprehensive reference for all of the commands and definitions in the system.

**18.2.1. System Section Keyword Listing***Dos Close Standard Files*

Close all files before running an external program. This feature should normally be enabled, since it allows external programs to write to the system log file and manipulate the area data files.

*File Access <file >*

This is the root name of the privilege level database. This keyword must come below the **File Password** keyword. SILT will create the access data file in the Maximus system directory when run with the -p or -x switches.

*File Callers <rootname>*

Name of the optional caller log file. This is a binary log that can be read by third-party utilities (and by using the **call\_open** and related MEX functions). When the user is logged off, a 128 byte record is appended to this log file.

*File Password <filespec>*

This keyword tells Maximus where to find the system user file. The **<filespec>** parameter must not have an extension. This file contains log-on and password information for all users on the system. If you are just starting out with Maximus and do not yet have a user file, start Maximus using the “-c” command line parameter.

*Log File <log\_name>*

This keyword tells Maximus what to call your system log file. The system log file is a human-readable version of the **File Callers** file. The log describes *who* called your system, *when* they called, and *what* they did while on-line. This statement can be overridden from the command line with the -l parameter.

*Log Mode <log\_type>*

This keyword tells Maximus how much information to place in the system log file. Maximus supports three logging modes for **<log\_type>**, as shown below in Table 18.1:

**Table 18.1 Logging Modes**

| Type    | Description  |
|---------|--|
| Trace   | Maximus logs all events.   |
| Verbose | Maximus logs most system events, including file transfers and external programs. |



---

Terse      Maximus only logs major system events, such as users logging off and errors.

---

*MCP Pipe <path>*

**OS/2 only!**

The **<path>** parameter specifies the pipe name to use for the MCP server process. The -a command line parameter can be used to override this value.

*MCP Sessions <num>*

**OS/2 only!**

This keyword sets an upper limit on the maximum number of Maximus sessions that will be running at the same time. Maximus passes this information to MCP when it starts up. (MCP uses **<num>** to determine how many named pipes to create.)

*Multitasker <type>*

**DOS only!**

This keyword is only required if you are running Maximus under a multitasking environment. **<type>** can be any of the following:

- ◆ DESQview
- ◆ DoubleDOS
- ◆ PC-MOS
- ◆ MSWindows
- ◆ None

*Name <bbs\_name>*

This keyword contains the name of your BBS. This name will be shown in a number of **.bbs** display files and on the “\* Origin:” line for EchoMail areas.

*No Password Encryption*

This flag disables the default password encryption feature. Enabling this keyword will not decrypt any currently-encrypted passwords in the user file, but it will prevent Maximus from automatically encrypting the passwords for new users. (Maximus will still be able to handle existing entries from the user file that contain encrypted passwords.)

*No SHARE.EXE*

This keyword tells Maximus that you are running MS-DOS version 3.3 (or prior) and that you do not have **share.exe** loaded. This keyword should only be used as a last resort, since data file corruption may occur if you try to run in a multitasking environment without support for **share.exe**. **You are using this**

**keyword at your own risk. Lanius cannot provide support for systems that run with this keyword enabled.**

*Path Inbound <path>*

This path tells Maximus where to find the files that are received by a front end mailer. Maximus will use this directory if you turn on FTS-0001 file attach support by enabling the **Style Attach** attribute in a NetMail message area.



If you make this facility available to non-trusted users, you may compromise the security of files in your inbound directory. Regular local file attach areas are safe for all users, but due to the nature of FTS-0001 attaches, you should only allow trusted users to access a NetMail area that has the local file attach feature enabled.

*Path IPC <path>*

**DOS only!**

This keyword is only required for DOS users. Maximus-OS/2 automatically uses the MCP communications server to handle its inter-process communications needs.

This keyword defines the path for the inter-process communications area. Maximus will use this directory to store temporary files for the multi-node chat facility. This keyword must point to an empty directory that you are not using for any other purpose.

You must have **share.exe** loaded to use the **Path IPC** feature.

*Path Language <path>*

This keyword tells Maximus where to find your system language files.

*Path Misc <path>*

This keyword tells Maximus where to find miscellaneous system display files. Aside from many of the standard **.bbs** files, this is also where the function-key files (**f\*.bbs**, **cf\*.bbs**, **af\*.bbs**, and **sf\*.bbs**) are located. Please see Appendix D for more information on these files.

*Path Outbound <path>*

This keyword is obsolete.

*Path System <full\_path>*

This keyword tells Maximus where to find the main system files. The **<full\_path>** must be a fully-qualified file specification, including the drive specifier and leading backslash. For example, two valid paths are:

```
C:\Max
D:\Bbs\Maximus
```

*Path Temp <path>*

This keyword tells Maximus the name of a directory that it can use for storing temporary files. Maximus will routinely delete files from this directory, so you must not store anything else here.

*Reboot***DOS only!**

This keyword tells Maximus to activate the FOSSIL “watchdog” feature. This means that the FOSSIL will reboot the computer if a user drops carrier while running an external program.

*Snoop*

This keyword controls the default state of the “Snoop” setting. When Snoop is enabled, the local console will display exactly what is shown to the remote caller. If Snoop is disabled, the local screen will contain just a status line display, similar to the system log file. Disabling Snoop will increase performance on a multi-node system.

Snoop can be manually enabled from the SysOp console using the “N” key.

*Swap***DOS only!**

This keyword instructs Maximus to swap itself out of memory before executing external programs. Maximus will automatically try to swap itself to XMS, EMS and disk (in that order) when running an external program. Maximus requires less than 2K of conventional memory when it is swapped out.

*SysOp <name>*

This keyword specifies the name of the system operator. This keyword does not automatically grant any special access rights; only those users with a privilege level high enough to put them in the SysOp user class are treated as SysOps.

However, this keyword controls the name that is displayed by the *[sysop\_name]* MECCA token and in the **To:** field in a **Leave\_Comment** message.

*Task* <task\_no>

This keyword specifies the default node number for the system. The node number can be overridden using the -n command line parameter.

If you are running a multinode system, ensure that all of your task numbers are non-zero. If you run a copy of Maximus with a node number of 0, it will not be able to communicate with the rest of the system.

*Video* <mode>

### DOS only!

This keyword only applies to DOS users. Maximus-OS/2 always uses **Video IBM**.

This keyword specifies the method to be used for local video display. Table 18.2 lists the video modes supported for <mode>:

**Table 18.2 Video Modes**

| Mode     | Description   |
|----------|---|
| BIOS     | Maximus writes all output to the screen using the video BIOS. This mode is almost as fast as Video IBM, but it will work on some PCs on which the IBM mode does not.                        |
| IBM      | Maximus writes directly to the video hardware. This is lightning-fast and is the quickest way to run Maximus, but it may not be compatible with some hardware or multitaskers.              |
| IBM/snow | This mode is identical to “Video IBM,” except that Maximus will wait for the vertical retrace before writing to the video buffer. This option may reduce flicker on some old IBM CGA cards. |

### 18.2.2. Equipment Section Keyword Listing

*Answer* <cmd>

In WFC mode, when Maximus detects a RING from the modem (as defined by the **Ring** keyword), it will send this modem command in response.

This command is used if you want Maximus to answer the phone itself, rather than trusting the modem hardware to pick up the line. For a standard Hayes modem, this keyword should read:

```
Answer ATA|
```

See the **Busy** keyword for special characters that can be used in the <cmd> string.

*Baud Maximum <speed>*

This keyword specifies the maximum speed that is supported by your modem. **<speed>** can be any of 300, 600, 1200, 2400, 4800, 9600, 19200, or 38400. Maximus-OS/2 also supports a speed of 57600.

*Busy <cmd>*

This keyword specifies the string that Maximus sends to the modem after a caller logs off.

Table 18.3 lists the characters in the **<cmd>** string that Maximus translates into certain actions:

**Table 18.3 Modem Translation Characters**

| Character | Description                  |
|-----------|------------------------------|
| v         | Set DTR low                  |
| ^         | Set DTR high                 |
| ~         | Pause for one second         |
| `         | Pause for 1/20th of a second |
|           | Send a carriage return       |

*Connect <string>*

This keyword specifies the string that is returned by the modem when a connection is successfully established. To determine the speed of the caller, Maximus will parse the text that the modem sends after this string.

*Init <cmd>*

This keyword specifies the string that Maximus sends when it needs to initialize the modem. Please see the **Busy** keyword for information on special characters that can appear in this string.

A normal **Init** string looks like this:

```
Init ~v~` `` ` | ~^ `` ATH0 |
```

If you want the modem to answer the phone automatically (and if you have commented out the **Answer** string), use this string:

```
Init ~v~` `` ` | ~^ `` ATH0S0=1 |
```

If you want Maximus to answer the phone itself, use the first **Init** string and enable the **Answer** keyword.

*Mask Carrier <mask>***DOS only!**

This keyword tells Maximus which Modem Status Register (MSR) bit to use for detecting on-line callers. This value is 128 for all modems made for use in North America and Europe, and it is also 128 for the vast majority of modems in all other countries. Do not change this value unless your modem's manual specifically instructs you to do so.

*Mask Handshaking <flow\_control>*

This keyword tells Maximus to enable or disable various types of flow control. The supported types for **<flow\_control>** are:

- ◆ XON
- ◆ CTS
- ◆ DSR

You must enable **XON** flow control if you want your users to be able to use **<ctrl-s>** and **<ctrl-q>** to start and stop screen display.

You must enable **CTS** flow control if you are using a high speed modem. Some (but not all) modems also support **DSR** flow control.

*No Critical Handler***DOS only!**

Use this command to disable the internal critical error handler. The critical error handler instructs DOS to fail the operations that would normally halt the system with an "Abort, Retry, Fail?" error message. Instead, it will print one of the following error messages:

```
Critical error reading/writing drive X:
Critical error accessing device COMx
```

*Output <port>*

This keyword sets the default COM port for modem output.

**DOS only!**

The DOS version of Maximus supports **<port>** values of "Com1" through "Com8" and "Local".

**OS/2 only!**

The OS/2 version of Maximus provides support for "Com1" up to the highest COM port that you have installed, in addition to the "Local" port.

For either operating system, when using ports higher than COM2, note that your FOSSIL (or OS/2 communications driver) and modem hardware must also specifically support the port that you wish to use. Check with your FOSSIL or driver manuals for details on this.

This setting can be overridden from the command line. The `-p` command line parameter instructs Maximus to use an alternate COM port, while the `-k` parameter instructs Maximus to log on locally.

#### *Ring <string>*

This keyword specifies the response sent by your modem when the phone is ringing. When Maximus receives this string, it will send the string specified by the **Answer** keyword.

#### *Send Break to Clear Buffer*

This keyword can only be used by owners of U.S. Robotics Courier or Sportster modems.

This keyword instructs Maximus to send a BREAK signal whenever it needs to clear the transmit buffer. This feature makes hotkeys slightly more responsive.

To enable this feature, you **must** include the “&Y0” parameter in your USR modem’s initialization string. Otherwise, the modem may transmit the BREAK signal to the remote modem instead of clearing its transmit buffer.

At the time of writing, only the U.S. Robotics modems implement this feature correctly. Do not try to use **Send Break to Clear Buffer** with modems from other manufacturers.

### **18.2.3. Matrix/EchoMail Keyword Listing**

#### *Address [zone:]<net/node>[.point]*

This keyword specifies the FidoNet-compatible network address used by Maximus. This address is inserted in message headers when writing NetMail or EchoMail messages. You may specify up to 16 addresses, but the first address will be used by default.

If you are running Maximus in a point configuration, the first **Address** line must specify your full 4D point address. The second **Address** line should specify your “fakenet” address. If you have no fakenet address, the second **Address** line should specify your host’s address.

#### *After EchoMail Exit <errorlevel>*

This keyword tells Maximus which errorlevel to use when a user enters EchoMail messages. This errorlevel will supersede the **After Edit** errorlevel.

*After Edit Exit* <errorlevel>

This keyword tells Maximus which errorlevel to use when a user enters NetMail messages. This errorlevel is superseded by the **After EchoMail** errorlevel.

*After Local Exit* <errorlevel>

This keyword tells Maximus which errorlevel to use when a user enters local messages. This errorlevel is superseded by both the **After EchoMail** and **After Edit** errorlevels.

*FidoUser* <filespec>

This keyword specifies the name and location of a user/address list (in standard **fidouser.lst** format). This file can be used for performing automatic address lookups when writing NetMail messages.

The list must be stored as a sorted ASCII text file. Each line must be exactly 60 columns wide in the format shown below:

|            |           |
|------------|-----------|
| Davis, Bob | 1:106/114 |
| Doe, Jane  | 1:13/42   |
| Doe, John  | 1:1/2     |
| Dudley, S  | 1:249/106 |

*Gate NetMail*

This keyword instructs Maximus to “gate route” interzone NetMail through the FidoNet zone gate. This function is handled by the Squish mail processor, so this keyword should normally be left commented out.

*Log EchoMail* <filespec>

This keyword instructs Maximus to create a log of EchoMail messages entered by the user. If this keyword is enabled and a **Tag** keyword is specified for a message area, Maximus will write the tag for that message area to <filespec> when the caller logs off.

This file can then be used by an external mail processor (such as Squish) to scan the specified areas for outgoing mail.

*Message Edit* <action> <attribute> <priv>

The **Message Edit** keyword series tells Maximus what to do when a user enters a NetMail message.



**<action>** can be one of “Ask” or “Assume.” For **Ask**, Maximus will ask the user whether or not the specified attribute should be set. For **Assume**, Maximus will automatically set the attribute without asking the user.

If the full-screen reader is enabled, **<action>** has a slightly different functionality. Attributes marked with **Assume** will be forced on when the user enters a message. Maximus will not explicitly prompt the user for an attribute when **Ask** is used, but it will allow the user to select that attribute from the message attribute field (above the message date).

Valid values for **<attribute>** are:

- ◆ Private
- ◆ Crash
- ◆ FileAttach
- ◆ KillSent
- ◆ Hold
- ◆ FromFile
- ◆ FileReq
- ◆ UpdateReq
- ◆ LocalAttach

#### *Message Send Unlisted <priv> <cost>*

This keyword controls the privilege level required to send NetMail to nodes which are not in the system nodelist. If the user’s privilege level is less than **<priv>**, Maximus will not allow the user to send the message. If the user’s privilege level is at least **<priv>**, Maximus will charge the user **<cost>** cents for sending the message.

#### *Message Show <item> to <priv>*

This keyword tells Maximus to display message control information to certain groups of users.

**<item>** can be any of the items in Table 18.4 below. To be able to view the specified item, a user must have a privilege level of at least **<priv>**.

**Table 18.4 Message Show Items**

| Item    | Description  |
|---------|--|
| Ctl_A   | This allows the user to view the <i>&lt;ctrl-a&gt;</i> “kludge lines” in EchoMail messages. This ability can also be toggled using the <b>Msg_Kludges</b> menu option. |
| Seenby  | This allows the user to view the “SEEN-BY” information at the bottom of an EchoMail message.   |
| Private | This allows the user to read private mail, regardless of the mes-  |

sage's addressee. This ability is normally only granted to SysOps. (Normally, users can only see private messages which are to or from themselves.)

---

#### *Nodelist Version <version>*

This keyword specifies the FidoNet nodelist format to use when sending NetMail messages.

<version> can be any of the following formats:

- ◆ 5
- ◆ 6
- ◆ 7
- ◆ FD (FrontDoor)

The first three options specify the "Version5," "Version6" and "Version7" nodelist formats (respectively). The "FD" option specifies the FrontDoor nodelist format. Maximus expects to find the nodelist files in the directory specified by **Path NetInfo**.

However, Maximus can send NetMail even without a nodelist; to do so, you must first uncomment the **Message Send Unlisted** keyword and set the privilege level appropriately.

Next, go to directory specified by **Path NetInfo** and create a zero-length file called **nodelist.idx**. The following DOS command can be used to accomplish this task:

```
REM > NODELIST.IDX
```

#### **OS/2 only!**

The above command only works from a DOS session. The OS/2 command interpreter will not create 0-length files, so you must go to DOS to execute this command.

Once this has been done, Maximus will be able to send and receive NetMail without a nodelist.

#### *Path NetInfo <path>*

This keyword tells Maximus where to find your system nodelist files.

#### 18.2.4. Session Section Keyword Listing

##### *After Call Exit <errorlevel>*

This keyword tells Maximus which errorlevel to use after a caller logs off (without leaving any messages).

##### *Alias System*

This keyword tells Maximus to display the user's alias in situations where it would normally display the user's name. This setting applies to user lists, to the multinode chat facility, and in message areas. (However, you can explicitly force the use of real names in a message area using the **Style RealName** keyword.)

The **Ask Alias** keyword can also be used in conjunction with **Alias System** to prompt new users for an alias at log-on.

Table 18.5 describes the various combinations of the two keywords:

**Table 18.5 Ask Alias and Alias System Options**

|                     |            | <b>Ask Alias</b>  |   |
|---------------------|------------|---|---|
|                     |            | <b>Yes</b>  | <b>No</b>   |
| <b>Alias System</b> | <b>Yes</b> | New users are prompted for an alias at log-on. By default, messages entered by users will use the alias unless <b>Style RealName</b> is specified for that area. Users show up in <b>Who Is On</b> as their aliases. The alias field is searched and displayed when doing a <b>Userlist</b> .                     | New users are not prompted for an alias at log-on. If the user selects an alias in the Change Setup section, this setting causes Maximus to function as if both <b>Ask Alias</b> and <b>Alias System</b> were enabled in the system control file. |
|                     | <b>No</b>  | New users are prompted for an alias at log-on. By default, messages entered will use the user's real name unless <b>Style Alias</b> is used in the message area definition. Users show up as their real names in <b>Who is On</b> . The user's real name is searched and displayed when doing a <b>Userlist</b> . | No alias use whatsoever.  |

*Area Change Keys <keylist>*

This keyword tells Maximus which keys to support on the **Area Change** menu for message and file areas. The first key in **<keylist>** is used to select the prior area. The second key in **<keylist>** is used to select the next area. The third key in **<keylist>** is used to select the area menu.

In addition to the keys specified here, the user will always be able to use the “[”, “]” and “?” keys to select the prior area, next area, and area list (respectively).

*Ask Phone*

This keyword tells Maximus to prompt new users to enter a phone number at log-on. This information is stored in the user file.

*Ask Alias*

This keyword tells Maximus to prompt new users to enter an alias at log-on. See the **Alias System** keyword for more details.

*Attach Archiver <arctype>*

This keyword specifies the storage type to use for compressed file attaches. The archiver specified by **<arctype>** must be a valid archiver defined in **compress.cfg**.

*Attach Base <root>*

This keyword specifies the root name of the file attach database. This keyword should specify only the path and up to four letters for the root of the database name. A number of additional database files are created using the root name.

For example, given the following statement:

```
Attach Base C:\Max\Att
```

Maximus will create the following database files:

```
C:\Max\Attmsg.db
C:\Max\Attname.i00
C:\Max\Attfile.i01
C:\Max\Attfile.i02
```

*Attach Path <path>*

This keyword specifies the default holding location for local file attaches. Unless otherwise specified, local file attaches will be stored in this area. This directory can be overridden on an area-by-area basis using the **AttachPath** keyword in the message area control file.

*Charset Swedish*

Enable internal support for the Swedish 7-bit character set. You must also edit **english.mad** and modify the LBRACKET and RBRACKET definitions to provide proper support for this character set.

*Charset Chinese*

Enable internal support for the Chinese character set. The Chinese character set provides for the “BIG5” two-byte codes used by most Chinese programs.

*Chat Capture On*

This keyword instructs Maximus to automatically enable the chat capture buffer when entering SysOp chat mode, instead of forcing the SysOp to press *<alt-c>* to manually enable the buffer. This feature does not work when an external chat program is enabled.

*Chat External <prog\_name>*

This keyword tells Maximus to use an external program for chatting instead of using the internal chat routine. Maximus will run **<prog\_name>** instead of the internal chat program when the SysOp presses *<alt-c>* on the local console.

To use a MEX program as an “external” chat program, specify a “:” in front of the name of the program. For example:

```
Chat External      :M\Mexchat
```

*Check ANSI*

This keyword instructs Maximus to query the remote terminal about ANSI support whenever a user logs on with ANSI graphics enabled. If the user’s terminal does not report that it supports ANSI, Maximus will prompt the user to disable ANSI graphics.

This keyword is disabled by default. A number of ANSI-supporting terminal programs do not support the query that Maximus uses in this check, and on occasion, Maximus may assume that a true ANSI-compatible terminal does not support ANSI.

*Check RIP*

This keyword instructs Maximus to query the remote terminal about *RIPscrip* graphics support. If the user's terminal program does not support *RIPscrip*, Maximus will prompt the user to disable *RIPscrip* graphics.

This keyword is enabled by default. All *RIPscrip* programs must support the "do you support RIP" query used by Maximus, so this keyword helps to ensure that Maximus only sends *RIPscrip* graphics to terminal programs that support it.

*Comment Area <area>*

This keyword tells Maximus where to place log-off message left by users. This area will also be used for messages created with the *[leave\_comment]* MECCA token.

*Edit Disable <option>*

This keyword tells Maximus to disable certain editor-related options.

**<option>** must be one of the following:

- ◆ MaxEd
- ◆ UserList

Selecting **MaxEd** tells Maximus to disable the MaxEd editor. All users will be forced to use the line editor.

Selecting **UserList** tells Maximus to disable the "user list" feature when leaving private messages in local message areas. (By default, users can enter "?" at the **To:** prompt to get a list of users on the system.)

*FileData <name>*

This keyword tells Maximus the root filename to use when creating the file area data file. Maximus stores all information related to file areas in files named **<name>.dat** and **<name>.idx**.

*File Date <type> [format]*

This keyword tells Maximus how to display dates to the user. Maximus supports a number of data formats, including U.S., Canadian/British, Japanese and Scientific.

In addition, Maximus can be instructed to retrieve file dates and sizes directly from the directory entry, or it can obtain that information from the file list itself.

If **<type>** is **Automatic**, Maximus will look at the file's directory entry to determine the file's size and date.

If **<type>** is **Manual**, Maximus will assume that the size and date information is stored as part of the comment in the **files.bbs** listing for the file area.

The automatic and manual dates can be overridden on an area-by-area basis using the **Type DateAuto**, **Type DateList** and **Type DateManual** keywords.

**[format]** is the optional format to use when displaying file dates. It can be any of the following options:

```
mm-dd-yy (U.S. - default)
dd-mm-yy (Canadian/British)
yy-mm-dd (Japanese)
yyymmdd  (Scientific)
```

If **<type>** is **Automatic**, the format above will be used when displaying the dates from the file directory.

If **<type>** is **Manual**, Maximus will use the above date format when inserting dated entries in the **files.bbs** catalog for uploads. You must manually insert dates for any preexisting files in the file areas.

The format specified by **[format]** is also used when prompting the user for a date during a new files scan.

Examples:

```
File Date Automatic dd-mm-yy
```

This line tells Maximus to automatically determine the dates and sizes of files from their directory entries, and also to display dates using a Canadian format.

```
File Date Manual yy-mm-dd
```

This line tells Maximus to expect to find file sizes and dates inserted directly into **files.bbs**, and when inserting upload descriptions into the file catalog, to insert dates using the Japanese date format.

*FileList Margin <col>*

This keyword instructs Maximus to indent long **files.bbs** descriptions by a particular offset for the second and subsequent lines. This may be useful if an external program is used to add a “download counter” to file descriptions; the margin can be increased to create a hanging indent, thereby placing the download counter in a separate column.

*First File Area <area>*

New users will be placed in the file area specified by **<area>** when they first log on. All users must be able to access at least one file area at all times.

*First Menu <menu>*

This keyword defines the name of the menu to be shown after Maximus finishes displaying **welcome.bbs**.

*First Message Area <area>*

New users will be placed in the message area specified by **<area>** when they first log on. All users must be able to access at least one file area at all times.

*Format Date <date\_format>*

The **Format Date** keyword controls the format of dates displayed by Maximus.

This format is used to display dates in message headers and in various other places throughout the system. The date format is output to the user exactly as specified, except for several special translation characters.

Table 18.6 describes the case-sensitive date translation characters supported by Maximus:

**Table 18.6 Date Format Characters**

| Format | Description                              |
|--------|--|
| %A     | Either “am” or “pm,” as appropriate.     |
| %B     | The month (in decimal).                  |
| %C     | The month as an abbreviated text string. |
| %D     | The day-of-month (in decimal).           |
| %E     | The hour (in the range of 1 to 12).      |
| %H     | The hour (in the range of 0 to 23).      |
| %M     | The minute.                              |
| %S     | The second.                              |
| %Y     | The year (without the century).          |
| %%     | A single percent sign.                   |



Examples:

`%E : %M %A`

This translates to the time in a 12-hour format. An example time shown with this format would be “08:23pm”.

`%H : %M : %S`

This translates to the time in a 24-hour format, including seconds. An example time shown with this format would be “20:23:15”.

`%B - %D - %Y`

This translates to the current date in a numeric format. An example date shown with this format would be “07-16-95”.

`%D %C %Y`

This translates to the current date in mixed format, with a numeric day-of-month, an abbreviated month, and a numeric year. An example date shown with this format would be “01 Aug 95.”

*Format FileFormat <format>*

*Format FileHeader <format>*

*Format FileFooter <format>*

*Format MsgFormat <format>*

*Format MsgHeader <format>*

*Format MsgFooter <format>*

These keywords tell Maximus how to display the message and file area lists that are generated when no custom **Uses MsgAreas** or **Uses FileAreas** files are defined.

When displaying a standard message or file area listing, the **MsgHeader** (or the **FileHeader**, as appropriate) is displayed at the top of the screen. Next, for each area to be displayed, one copy of the **MsgFormat** or **FileFormat** string is displayed. Finally, after displaying all of the areas, Maximus displays the **MsgFooter** or **FileFooter** string.

These strings can contain a number of optional formatting characters in addition to standard ASCII text. The percent sign (“%”) is used as an initiator for the formatting sequences. Anything not preceded by a percent sign is passed directly on to the user.

The format of each control sequence is given below:

```
%[-][min][max<format_char>
```

Everything except for **<format\_char>** is optional and may be omitted.

The optional “-”, if present, specifies that the output of this formatting character must be left-justified.

The optional **min** value specifies the minimum width for the output of this formatting character. When used in conjunction with the “-” character, the output will be left-justified. Without the “-” character, the field will be right-justified.

The optional **max** value specifies the maximum width for the output of this formatting character. If the data field is longer than **max**, the field is truncated. The period (“.”) in front of the **max** value is not optional.

**format\_char** specifies the type of output to display. This character is case-sensitive. The supported characters are listed in Table 18.7:

**Table 18.7 MsgFormat/FileFormat Characters**

| Character | Description  |
|-----------|--|
| #         | Display the name of the current area.  |
| *         | In a <b>MsgFormat</b> statement, this character displays an asterisk (“*”) if the user has unread messages in the specified area. Otherwise, this character displays a space (“ ”). (When displaying a list of areas from the <b>Msg_Tag</b> menu option, this token is also used to indicate whether or not a message area is tagged.)                        |
| a         | The name of the current area with the division prefix removed.   |
| d         | The name of the current division.  |
| c         | When specified in the format “%#.\$c”, Maximus will skip the next “\$” characters in the <b>MsgFormat</b> or <b>FileFormat</b> sequence after every “#”th area is processed. (For example, the sequence “%2.3cABC” instructs Maximus to display the sequence “ABC” for every second area.)   |
| f         | Display a <b>.bbs</b> file. The name of the file to display must immediately follow the “%f” token. The name of the file must be separated from the remainder of the string with a space. (When instructed to display a file, Maximus will always display that file before showing any of the other text in the <b>MsgFormat</b> or <b>FileFormat</b> string.) |
| n         | Display the area’s description (from the <b>Desc</b> keyword in the area definition).  |
| t         | Display the tag for a message area, as specified in the <b>Tag</b> keyword in the area definition.   |
| x         | Display the next two characters in the sequence as a single character. The following two characters must be hexadecimal digits that represent a single character.  |

Examples:

```
%-15.15t
```

This translates into the “echo tag” for the current message area, left justified and exactly fifteen characters long (padded with spaces).

```
%30.30n
```

This translates into the name of the current message area, right justified to make the field exactly 30 characters long.

```
%%*%2# / %-25.25n %2c%x0a
```

This statement causes Maximus to display the area number for each area, followed by a space and a forward slash, another space, and the area name (left-justified to a maximum length of 25 characters). After every second area, Maximus displays a linefeed (ASCII 12, or 0a in hexadecimal), effectively creating a two-column area display. Finally, if an area contained new messages, a “\*” would be displayed beside the area number.

#### *Format Time <format>*

The **Format Time** keyword controls the format of times displayed by Maximus.

This format is used to display times in message headers and in various other places throughout the system. The times format is output to the user exactly as specified, except for several special translation characters.

Please see the description of the **Format Date** keyword for information on the supported translation characters.

#### *Highest FileArea <area>*

This keyword specifies the highest file area number that can be viewed using the **File\_Locate** and **File\_Area** menu options. See also the **Type Hidden** keyword in the file area control file.

#### *Highest MsgArea <area>*

This keyword specifies the highest message area number that can be viewed using the **Msg\_Browse** and **Msg\_Area** menu options. See also the **Style Hidden** keyword in the message area control file.

*Input Timeout <mins>*

This tells Maximus to hang up on callers after **<mins>** minutes of inactivity. The default value is 4 minutes. This keyword can specify a range of 1 to 255 minutes.

*Kill Attach <type> [priv]*

This keyword determines how to handle automatic disposing of a file when a file attach is received.

**<type>** must be one of the keywords from Table 18.8:

**Table 18.8 Kill Attach Types**

| Type   | Description  |
|--------|--|
| Never  | An attached file is never deleted automatically. (The attached file may be deleted by manually deleting the message containing the attach.)  |
| Always | An attached file is always deleted after a successful download.  |
| Ask    | Ask the user if the file should be removed. If a privilege level is specified and the user's privilege level is less than this level, the user is not asked and the file is removed automatically. |

*Kill Private <when>*

This keyword controls Maximus's handling of private messages in local message areas.

**<when>** can be any of the values from Table 18.9:

**Table 18.9 Kill Private Types**

| Type   | Description   |
|--------|---|
| Always | Maximus always kills a private message after it has been read by the recipient. |
| Ask    | Maximus asks the user whether or not to kill a private message.                 |
| Never  | Maximus never kills a private message automatically.                            |

*Local Editor [!][@]<editor\_cmd>*

This keyword tells Maximus to use **<editor\_cmd>** as an external message editor for local users, rather than using the internal MaxEd editor.

With this keyword enabled, Maximus will execute **<editor\_cmd>** whenever a local user needs to compose a message. If the message being created is a reply to another message, Maximus will quote the original and place it in a file called

**msgtmp##.\$\$\$** before invoking the editor, where **##** is the current task number in hexadecimal.

After your editor returns, Maximus expects to find the final message in **msgtmp##.\$\$\$**, regardless of whether or not the message was a reply.

Normally, the **Local Editor** keyword only applies to users who are logged in at the local console. However, if the first character of the editor name is a “@”, certain remote users will also be allowed to use the “local” editor. See the **MailFlags Editor** and **MailFlags LocalEditor** keywords in the access control file for information on enabling this feature.

Finally, if you place the sequence **%s** inside the **<editor\_cmd>** string, Maximus will replace the “%s” with the name of the temporary file to be edited. This can be useful in a multitasking situation if you do not want to hard-code a task number in the control file.

#### *Local Input Timeout*

This statement instructs Maximus to apply the input timeout counter to local users in addition to remote users. (Normally, the input timer is disabled for local logons, meaning that that the SysOp can log on, go away for 30 minutes, and still have Maximus waiting at the input prompt.)

#### *Logon Level <priv>*

##### *Logon Preregistered*

This keyword specifies the privilege level to be assigned to new users. If you use the **Logon Preregistered** statement instead, new users will not be allowed to log on. (Maximus will instead hang up after displaying the **Uses Application** file.)

#### *Logon TimeLimit <time>*

This keyword specifies the maximum amount of time, in minutes, that the user is given to log on, read through the **Uses Application** file, enter a password, and so on.

#### *Mailchecker Kill <priv>*

##### *Mailchecker Reply <priv>*

These keywords control the privilege level required to access the **Kill** and **Reply** functions in the **Msg\_Browse** command and in the mailchecker.

The **Mailchecker Kill** keyword controls the privilege level required to delete a message using the Browse command.

|                        |
|------------------------|
| Title: Aff<br>Creator: |
|------------------------|

The **Mailchecker Reply** keyword controls the privilege level required to reply to a message using the Browse command.

When Maximus determines whether or not a user has access to the **Reply** and **Kill** commands, it also examines the message menu to find the privilege levels for the **Msg\_Reply**, **Msg\_Kill** and **Msg\_Upload** options.

For this feature to work properly, you must have a menu explicitly called “MESSAGE”, and that menu must contain options for **Msg\_Reply**, **Msg\_Kill** and **Msg\_Upload** (with privilege levels that can be accessed by the user).

Even if you have renamed your main message menu to something else, you still need a menu called “MESSAGE”. (Note that this menu need not be reachable from your normal menu structure. The sole purpose of this menu is to store the privilege levels required to access the **Reply** and **Kill** commands in the mail-checker.)

#### *MaxMsgSize <size>*

This keyword controls the maximum size of messages uploaded using the **Msg\_Upload** menu option.

The value for **MaxMsgSize** does not apply to messages entered locally, nor does it apply to users in a class with the **MailFlags MsgAttrAny** flag.

#### *Menu Path <path>*

This keyword tells Maximus where to find the default menu (\*.mnu) files

#### *MessageData <file>*

This keyword tells Maximus the root filename to use when creating the message area data file. Maximus stores all information related to message areas in files named <name>.dat and <name>.idx.

#### *Min Logon Baud <speed>*

This keyword specifies the minimum speed at which a user must call in order to log on to the system. See also the **LogonBaud** keyword in the access control file.

#### *Min NonTTY Baud <speed>*

This keyword specifies the minimum speed at which a user must call in order to use a terminal mode other than TTY.

*Min RIP Baud <speed>*

This keyword specifies the minimum speed at which a user must call in order to use *RIPscrip* graphics. *RIPscrip* support can be disabled by setting this value higher than the maximum baud rate.

A new user must be logged in at a speed that is at least equal to or higher than the **Min NonTTY Baud** and **Min RIP Baud** speeds in order to enable *RIPscrip*. Maximus attempts to auto-detect *RIPscrip* capability on the remote and will set the default response in the new user logon sequence accordingly.

*No RealName Kludge*

This keyword instructs Maximus not to insert the *^aREALNAME:* line into messages entered in an anonymous message area.

Normally, this line aids in tracking down users who try to abuse the ability to leave anonymous messages. However, there are circumstances when you want to ensure that a user can leave messages which are completely confidential, and enabling this keyword will do that. This option can be overridden on an area-by-area basis; see the **Style** keyword in the message area control file for more information.

*RIP Path <path>*

This keyword tells Maximus where to find all of the default **\*.rip** and **\*.icn** files to be sent using the *[ripdisplay]* and *[ripsend]* MECCA tokens.

This path can be changed at runtime using the *[rippath]* MECCA token.

*Save Directories <drives>***DOS only!**

When Maximus runs an external program, this keyword instructs it to record the “current directory” for the specified drives.

The **<drives>** parameter is a list of all drive letters on your system, except for removable media (floppy drives and CD-ROMs).

*Single Word Names*

This keyword instructs Maximus to permit usernames that contain only a single word. This option may be useful on systems that support aliases. (Maximus will also suppress the default “What is your LAST name:” prompt when this keyword is enabled.)

*Stage Path <path>*

This keyword tells Maximus to use **<path>** as a temporary directory for staging file transfers.

This path is only used when a file area is declared using **Type Staged** or **Type Hidden**. In areas with this type, Maximus will copy the files from the CD-ROM to the specified staging directory before sending the files. This reduces wear and tear on normal CD-ROMs, and it prevents thrashing for multi-disk CD changers.

The declared path should always have at least as much space as allowed for in your daily download limits. (If Maximus is unable to copy a file to the staging area, the transfer will send the file from its original location.)

*StatusLine*

This keyword instructs Maximus to display a status line at the bottom of the screen. The status line is only active for remote users.

*Strict Time Limit*

This keyword instructs Maximus to terminate callers who run over their time limits while in the middle of a download. If this happens, Maximus will create a log entry and abort the file transfer. This feature only works for internal protocols.

*Track Base <root>*

This keyword specifies the path and root filename of the MTS database. This keyword should only specify the path and up to four letters for the root of the database name.

For example, assuming the following line:

```
Track Base c:\max\trk
```

Maximus will create the following database files:

```
c:\max\trkmsg.db
c:\max\trkmsg.i00
c:\max\trkmsg.i01
c:\max\trkmsg.i02
c:\msg\trkarea.i00
c:\msg\trkown.i00
```



*Track Exclude <filespec>*

This keyword gives the name of the “exclusion list” for controlling automatic message assignment in MTS areas.

In some situations, a MTS moderator may not wish to track messages that are entered by certain users. The exclusion list is a flat text file, one name per line, which lists the names of all users to be excluded from the tracking system.

Messages entered by these users can be manually placed in the tracking system using the **Track/Insert** command.

*Track Modify <priv>*

This keyword controls the privilege level required to modify tracking information for messages which are not owned by the user performing the modification. (In this case, “owned” refers to the user in the “owner” field of that message in the MTS database.)

This privilege level is also required to delete messages from the tracking database and to access the **Track/Admin** menu.

*Track View <priv>*

This keyword controls the privilege level required to view tracking information of messages owned by others. To view tracking information in a message, that message must either be owned by the current user, or that user must have a privilege level of at least the value specified by this keyword.

*Upload Check Dupe**Upload Check Dupe Extension*

These keywords instruct Maximus to check for duplicate files when processing uploads.

The **Upload Check Dupe** keyword tells Maximus to compare only the root part of the filename. Using this method, “foo.zip” and “foo.lzh” will be considered duplicates.

The **Upload Check Dupe Extension** keyword tells Maximus to compare the full filename and extension of the uploaded file.

To use this feature, you must use the FB utility to create a file database.

*Upload Check Virus <batchfile>*

This keyword instructs Maximus to call a batch or command file every time a file is uploaded.

<**batchfile**> specifies the name of the batch file to run. For example:

```
Upload Check Virus    vircheck.bat
```

(For OS/2, use **vircheck.cmd**.)

This keyword tells Maximus to call **vircheck.bat** every time a file is uploaded. Maximus will call the batch file using the following format:

```
vircheck.bat path name extension miscdir task
```

**path** is the path of the uploaded file (including the trailing backslash).

**name** is the root name of the file (without the extension).

**extension** is the extension of the file, beginning with a period (“.”).

**miscdir** is the path to the Maximus **\max\misc** directory.

**task** is the current node number.

Please note that the file name and extension are passed as two separate arguments. This allows the batch file to easily check for uploads with a certain extension.

The batch file can process the uploads as desired, including scanning for viruses, refusing files with bad extensions, and so on. After the batch file returns, Maximus will check again to see if the uploaded file still exists.

If the file still exists, Maximus displays **\max\misc\file\_ok.bbs**. Normally, this file contains a message informing the user that the file contained no viruses. Maximus will then ask for an upload description and credit the user's account.

If the uploaded file no longer exists, presumably because it was removed by **vircheck.bat**, Maximus will display **\max\misc\file\_bad.bbs**. This file presumably mentions that the virus check failed.

This feature was designed for automated virus-checking programs, but other tricks can also be done with batch files. The uploaded file's extension can be tested as a separate argument, so it can be used to block uploads of files with certain extensions.

The `\max\misc\file_bad.bbs` file can also be swapped by `vircheck.bat` for another file, so a different file can be displayed for virus checks and archive corruption checks. The display file could also be used to display the log of the virus checking program, thereby giving the user more information about the virus itself.

*Upload Log <log\_name>*

This keyword specifies the name of a log file used for storing the names of uploaded files.

One line is written to this file for every upload received. Included is the name of the file uploaded, the name of the user who uploaded it, the size of the file, and the current date and time. This makes it very easy to keep track of which user uploaded which file.

*Upload Space Free <amount>*

This keyword tells Maximus to prevent users from uploading files if there are less than **<amount>** free kilobytes of space on the upload drive. (If there is less than **<amount>** kilobytes available, Maximus displays the **Uses NoSpace** file and refuses the upload.)

*Uses Application <filespec>*

This file is displayed to a new user after answering *Y* to the “Firstname Last-name [Y,n]?” prompt, but before prompting the user for city and phone number information.

*Uses BOREDhelp <filespec>*

This file is displayed to first-time callers who enter the BORED editor when their help level is set to *novice*.

*Uses BadLogon <filespec>*

This file is displayed to users who failed the last log-on attempt due to an invalid password.

*Uses Barricade <filespec>*

This file is displayed to users after they enter a barricaded message or file area, but before they are prompted for the password.

*Uses BeginChat <filespec>*

This file is displayed to the user when the SysOp enters CHAT mode. This is a good place to put something like, “Hi [user], this is the SysOp speaking.”

The default **Uses BeginChat** message is “CHAT: start”.

*Uses ByeBye <filespec>*

This file is displayed to users after they select the **Goodbye** menu option.

*Uses Cant\_Enter\_Area <filespec>*

This file is displayed to users when they try select an area that does not exist. The default response is “That area does not exist!”

*Uses Configure <filename>*

This file is displayed to new users after they log in, but before the standard user configuration questions are asked.

If the “configured” bit is set in the user record after this file is displayed (via a MEX script, for example), the standard configuration questions are skipped, thereby allowing the standard new user configuration to be replaced.

*Uses ContentsHelp <filespec>*

This file is displayed to users who request help for the **File\_Contents** command.

*Uses DayLimit <filespec>*

This file is displayed to users who try to log on after having overrun their daily time limits.

*Uses EndChat <filespec>*

This file is displayed to the user when the SysOp exits chat mode. The default response is “END CHAT.”

*Uses EntryHelp <file>*

This file is displayed to the user just before entering the message editor, regardless of whether the user is using the full screen editor or the line editor. This file can offer additional help or set up the screen display for *RIPscrip* callers.

*Uses FileAreas <filespec>*

This file is displayed to a user when a file area listing is requested. This display file replaces the automatically-generated file area list.

*Uses Filename\_Format <filespec>*

This file is displayed to users who try to upload files using an invalid filename.

*Uses HeaderHelp <file>*

This file is displayed to users just before the message header entry screen is displayed. This file can contain information regarding message attributes, using aliases, anonymous areas, and so on.

*Uses Leaving <filespec>*

This file is displayed just before Maximus exits to run an external program from a menu option.

*Uses LocateHelp <filespec>*

This file is displayed to users who request help using the **File\_Locate** command.

*Uses Logo <filespec>*

This file (normally called **\max\misc\logo.bbs**) is displayed immediately after Maximus connects with the user. This file should normally contain a small amount of information describing your BBS. This file must not contain ANSI or AVATAR graphics.

*Uses MaxEdHelp <filespec>*

This file is displayed to users who ask for help (by pressing “^k?”) from within the MaxEd editor.

*Uses MsgAreas <filespec>*

This file is displayed to a user when a file area listing is requested. If present, this file replaces the automatically-generated file area list.

*Uses NewUser1 <filespec>*

This file is displayed to a new user right before Maximus asks the user to enter a password.

*Uses NewUser2 <filespec>*

This file is displayed to a new user in lieu of the **Uses Welcome** file.

*Uses NoMail <filespec>*

This file is displayed to a user after the *[msg\_checkmail]* MECCA token determines that there is no mail waiting for the user.

*Uses NoSpace <filespec>*

This file is displayed when the amount of space free on the upload drive is less than the value specified by the **Upload Space Free** keyword.

*Uses NotFound <filespec>*

This file is displayed to a new user after the user's name is entered, but before the "First Last [Y,n]?" prompt is displayed.

*Uses ProtocolDump <filespec>*

This file is displayed to the user instead of the standard "canned" list of protocol names. This file is displayed for both the **File\_Upload** and **File\_Download** menu options.

*Uses Quote <filespec>*

This keyword specifies the name of an ASCII text file that contain quotes and random pieces of wisdom. Each quote in the file should be separated by a single blank line. This file can be accessed using the MECCA *[quote]* token.

*Uses ReplaceHelp <filespec>*

This file is displayed to the user just after selecting the **Edit\_Edit** option on the editor menu. This file describes the search and replace feature of the line editor.

*Uses Returning <filespec>*

This file is displayed to the user upon returning from an external program invoked by a menu option.

*Uses Rookie <filespec>*

This file is displayed to a user who has called between two and eight times, in lieu of the **Uses Welcome** file.

*Uses Shell\_Leaving <filespec>*

This file is displayed to the user immediately after the SysOp presses <alt-j> on the local console to shell to the operating system.

*Uses Shell\_Returning <filespec>*

This file is displayed to the user after the SysOp returns from a shell.

*Uses TimeWarn <filespec>*

This file is displayed to the user just before displaying the main menu, as long as that user has made more than one call on the current day.

*Uses TooSlow <filespec>*

This file is displayed to users whose speed is lower than the minimum speed required in **Min Logon Baud**, or if the user's speed is less than the **Logon-Baud** keyword for the user's class in the access control file.

*Uses Tunes <filespec>*

This keyword specifies the name and location of the Maximus tunes file. This tune file can be used to play simple melodies on the PC speaker when a user yells. For more information on the format of this file, please see the comments in the distribution version of **tunes.bbs**.

*Uses Welcome <filespec>*

This file is displayed to normal users who have called more than eight times. This file is displayed immediately after the user enters the log-on password.

*Uses XferBaud <filespec>*

This file is displayed to users whose speed is less than the speed required for the **XferBaud** setting for their user class in the access control file.

*Use UMSGIDs*

This keyword instructs Maximus to use unique message identifiers when displaying messages in Squish areas.

This means that every message entered in a Squish area will have a unique message number assigned to it. This number will never be reused, regardless of message deletions. This feature only works for Squish-style bases, so this keyword has no impact on \*.MSG areas.

*Yell Off*

This keyword completely disables the **Yell** function.

## 18.3. Language Control File

### 18.3.1. Description

The default language control file is called **language.ctl**. This file defines the languages supported in Maximus's multilingual system. This file also allows the SysOp to select a default language by placing the **Language** keywords in a particular order.

### 18.3.2. Language Section Keyword Listing

*Language <filename>*

This keyword specifies the name of a Maximus-specific language file. For example, the statement "Language English" tells Maximus to look for a file called **english.ltf** in the **Path Language** directory.

Up to eight language files may be specified in this section. Under DOS, remember that language filenames must not be more than eight characters long.

The first language listed in this section is used as the default for both new users and the SysOp's log file.

## 18.4. Off-Line Reader Control File

### 18.4.1. Description

The default off-line reader control file is **reader.ctl**. If you do not wish to enable the off-line reader, the **Include Reader.Ctl** statement in **max.ctl** may be commented out.

### 18.4.2. Reader Section Keyword Listing

*Archivers <filespec>*

This keyword specifies the file that contains definitions for external compression utilities. The file is identical in format to the format of the **compress.cfg**



file used by the Squish mail processor. If you are running both Maximus and Squish, this keyword can point to the same compression file as used by Squish.

*Packet Name <filename>*

This keyword defines an eight-character identifier for your system. Maximus uses this identifier when building QWK packets. Downloaded packets will be called **<filename>.qwk**, and uploaded replies will be called **<filename>.rep**.

This keyword should normally specify an abbreviation of your BBS name. The abbreviation must be eight characters or less and cannot include spaces. Only valid DOS filename characters are permitted. (A-Z, 0-9, and !@#%&()\_.)

*Work Directory <path>*

This keyword specifies the name of the directory used for creating QWK packets. Maximus creates subdirectories under the path you specify for per-node storage.

However, any files contained in the main work directory (as specified by **<path>**) are included in all downloaded QWK packets.

*Max Messages <num>*

This keyword defines the maximum number of messages that Maximus will pack into one QWK packet. If you do not wish to limit the number of messages, specify **Max Messages 0**.

*Phone Number <phone number>*

This keyword defines the phone number of your BBS, as it should be shown in downloaded QWK mail packets. The number should be in the “(xxx) yyy-zzzz” format, since some off-line readers depend on the phone number looking like this. However, Maximus itself does not care about the format, so it will copy the string verbatim to the **control.dat** file in the QWK packet.

## 18.5. Colors Control File

### 18.5.1. Description

The default colors control file is **colors.ctl**. Maximus uses this file to select colors for certain Maximus prompts.

This control file contains only those colors which are shown to the SysOp. A large number of other colors are stored in the `\max\lang\colors.lh` language include file. Please see the comments inside that file for more information.

### 18.5.2. Colors Section Keyword Listing

#### *Popup Border <color>*

Color for the border of a pop-up window. The default color is yellow on blue.

#### *Popup Highlight <color>*

Color for highlighted text inside pop-up windows. The default color is yellow on blue.

#### *Popup LSelect <color>*

Color for selected items on pop-up pick lists. The default color is grey on red.

#### *Popup List <color>*

Color for standard items on pop-up pick lists. The default color is black on grey.

#### *Popup Text <color>*

Color for the text in a pop-up window. The default color is white on blue.

#### *Status Bar <color>*

Color for the main status bar. The default color is black on white.

#### *Status Chat <color>*

Color for the chat request indicator (“C”). The default color is blinking black on white.

#### *Status Key <color>*

Color for the rest of the status-line key flags, such as *K*. The default color is black on white.

#### *WFC Activity <color>*

Color for the activity window on the “Waiting for Caller” screen. The default color is white on blue.

*WFC ActivityBor <color>*

Color for the activity window border on the “Waiting for Caller” screen. The default color is lightcyan on blue.

*WFC Keys <color>*

Color for the keys window on the “Waiting for Caller” screen. The default color is yellow on blue.

*WFC KeysBor <color>*

Color for the keys window border on the “Waiting for Caller” screen. The default color is white on blue.

*WFC Modem <color>*

Color for the modem window on the “Waiting for Caller” screen. The default color is gray on blue.

*WFC ModemBor <color>*

Color for the modem window border on the “Waiting for Caller” screen. The default color is lightgreen on blue.

*WFC Line <color>*

Color for the bar at the top of the “Waiting for Caller” screen. The default color is white.

*WFC Name <color>*

Color for the Maximus name on the “Waiting for Caller” screen. The default color is yellow.

*WFC Status <color>*

Color for the status window on the “Waiting for Caller” screen. The default color is white on blue.

*WFC StatusBor <color>*

Color for the status window border on the “Waiting for Caller” screen. The default color is yellow on blue.

## 18.6. Message Area Control File

### 18.6.1 Description

The default message area control file is called **msgarea.ctl**. This file defines all of the message areas available on a Maximus system.

Every message area must begin with a **MsgArea** keyword, and every message area must end with a **End MsgArea** keyword.

### 18.6.2. Alphabetical Keyword Listing

#### *ACS*

This keyword specifies the access level required to see or access this message area.

#### *AttachPath <path>*

This keyword specifies the path to use for storing local file attaches that are created in this message area.

**<path>** should point to an empty directory that Maximus can use for storing the file attaches.

#### *Barricade <menu\_name> <barricade\_file>*

This keyword specifies a barricade file to be used for the current message area. As long as the current menu is **<menu\_name>**, the barricade privilege level from **<barricade\_file>** will override the user's normal privilege level. Please see section 4.6 for more information on barricade files.

#### *Desc <desc>*

#### *Description <desc>*

These keywords specify the description of the message area, as you wish it to appear on the message area menu.

#### *End MsgArea*

This keyword tells SILT that the message area definition is complete.

*MenuName* <orig\_menu> <replace\_menu>

While Maximus is in this message area, when it is instructed to display the menu <orig\_menu>, it will instead display <replace\_menu>.

*MsgArea* <area>

This keyword tells SILT to begin a message area definition. <area> is the “leaf” name of the message area; if the area is contained within a message division, the name of the division will be added to the front of the string when writing the area to the message area data file.

*MsgDivisionBegin* <name> <acs> <display\_file> <desc>

This keyword is used to begin a message area division. Divisions can be used to create a multi-level, hierarchical message area structure.

<name> is the name of the message area division. This should be a very short tag identifying the area type. (It should be short because it is added to the beginning of all area names declared within the division.)

<acs> is the access control string required to view the message division. Please note that the ACS specified for this division has nothing to do with the ACS required to enter a contained message area. (However, the ACS for the message areas in the division will typically be at least as restrictive as the ACS for the division.)

<display\_file> is the name of the message area display file to show when the user requests an area list of this division. This file is only used if the corresponding **Uses MsgAreas** statement is enabled in the system control file. If you are not using custom message area menus, specify a “.” as the filename.

<desc> is the description for this division, as it will appear on the message area menu.

*MsgDivisionEnd*

This keyword ends a message division.

*Origin* <primary\_addr> <seenby\_addr> [text]

For an EchoMail area, this keyword instructs Maximus to use an origin line other than the default origin in the system control file.

<primary\_addr> is the address to place in the **MSGID** kludge line at the top of the message. This address is also placed on the origin line.

<seenby\_addr> is the address to use in the **SEEN-BY** line.

A period (“.”) can be specified for either or both of <primary\_addr> and <seenby\_addr> to use the default address.

The optional [text] contains the new text to use on the origin line. If [text] is omitted, Maximus uses the default origin line text.

For example:

```
Origin . . New origin for this area
```

This statement simply changes the origin text.

```
Origin 1:249/106.4 1:24906/2 New point text
```

This statement changes the primary address to 1:249/106.4, changes the SEEN-BY address to 1:24906/2, and changes the origin line text to “New point text”.

*Override <menu\_name> <option\_type> <acs> [letter]*

The **Override** keyword instructs Maximus to alter the privilege level required to access a menu option while the user is in this message area.

<menu\_name> is the name of the menu containing the command to be overridden. In most cases, this will be “MESSAGE”.

<option\_type> is the menu option type to override. Any of the keywords from the menu control file can be used here. If you do not specify a [letter], the override will apply to all options on the menu with a type of <option\_type>.

<acs> is the new ACS required to access the menu option.

[letter] is the optional first letter of the command to be overridden. If this letter is specified, the override will only apply to menu commands which: are on the menu called <menu\_name>, which have an option type of <option\_type> and which have a first letter of [letter].

For example:

```
Override MESSAGE Msg_Reply SysOp/135 R
```

This command overrides the **Reply** option on the message menu. In the area where this definition is placed, a user must pass the ACS test of “SysOp/135” before being allowed to reply to a message in this area. The override is further restricted so that it only applies to commands that start with the letter “R”.

(Had there been other **Msg\_Reply** options on the menu that started with other letters, they would not be overridden.)

*Owner <id>*

This keyword sets a default MTS owner for this message area. The **Owner** keyword must be specified in every area that uses the **Style Audit** style. This owner/area link can also be modified using the **Track/Admin/Owner** menu.

*Path <path>*

This keyword tells Maximus where to find the files that store the data in this message area.

If the area uses the Squish format (default), **<path>** must specify the directory and root filename of the message files.

If this area uses the \*.MSG format, **<path>** must specify the name of the directory used for storing the message files. All \*.MSG areas must have a separate directory of their own; two \*.MSG areas cannot be stored in the same directory.

*Renum Days <num>*

This keyword tells Maximus to keep no more than **<num>** days worth of messages in this area.

For \*.MSG areas, messages are purged by the MR renumbering program.

For Squish areas, messages are purged using the SQPACK utility. (SQPACK is available as part of the Squish program.)

If you are using the Squish mail processor, you only need to set the renumbering option in one place — either in your message area control file or in your **squish.cfg** file.

*Renum Max <num>*

This keyword tells Maximus to store no more than **<num>** messages in the area at one time. For Squish-format areas, Maximus will automatically purge messages as new messages are added. For \*.MSG-format areas, the external MR utility must be run to purge messages.

If you are using the Squish mail processor, you only need to set the renumbering option in one place — either in your message area control file or in your **squish.cfg** file.

*Style <styles>*

This keyword sets an optional number of style flags for the current area. The style describes the types of messages that can be created, the physical message storage format, and various other options.

<styles> can be zero or more of the options from Table 18.10:

**Table 18.10 Message Area Styles**

| Style        | Description  |
|--------------|--|
| *.MSG        | Store the message area using the *.MSG message format. (The default is to store the area using Squish format.)   |
| Alias        | Use the user's alias in the message header, rather than the user's real name.  |
| Anon         | Allow anonymous messages. Users are allowed to modify the "From:" field of the message to change it to a name of their choosing. However, Maximus will still insert a kludge line in the message containing the user's real name. See the <b>NoNameKludge</b> style for information on disabling this feature. |
| Attach       | Enable local file attaches in this area. This style must be used in conjunction with <b>Style Squish</b> . Please see section 5.2 for more information.  |
| Audit        | Enable MTS tracking for this area. You also need to define the Owner keyword in this area to use the MTS feature. This style must also be used in conjunction with <b>Style Squish</b> . Please see section 5.6 for more information on the Message Tracking System.   |
| Conf         | This area is a conference area. Conference areas are similar to EchoMail areas, but conferences use PIDs and have no tear lines. This style cannot be used in conjunction with the <b>Net</b> , <b>Local</b> or <b>Echo</b> styles.  |
| Echo         | This area is an EchoMail area. This style cannot be used in conjunction with the <b>Conf</b> , <b>Net</b> or <b>Local</b> styles.  |
| HiBit        | Allow 8-bit IBM extended ASCII characters in messages entered in this area.  |
| Hidden       | This area should not be shown on the standard message area list. The <b>Msg_Area</b> prior and next commands will skip this area. However, the area can still be accessed by name.   |
| Loc or Local | This is a local message area. This style cannot be used in conjunction with the <b>Conf</b> , <b>Net</b> , or <b>Echo</b> styles.  |
| Net          | This is a NetMail area. This style cannot be used in conjunction with <b>Conf</b> , <b>Local</b> or <b>Echo</b> styles.  |
| NoMailCheck  | This area is skipped during the standard mail-checking routine. This style is useful for areas which never contain   |



|              |   |
|--------------|---|
|              | personal mail for users.  |
| NoNameKludge | This style toggles the <b>No RealName Kludge</b> setting in the system control file. This style is useful for preserving true anonymity in areas that accept “anonymous” messages.                                  |
| Pub          | Allow public messages in this area. If this flag is used in conjunction with <b>Style Pvt</b> , both public and private messages are allowed. If neither flag is specified, this area allows only public messages.  |
| Pvt          | Allow private messages in this area. If this flag is used in conjunction with <b>Style Pub</b> , both public and private messages are allowed. If neither flag is specified, this area allows only public messages. |
| ReadOnly     | Only users in a class with <b>MailFlags WriteRdOnly</b> are allowed to write messages in this area.   |
| RealName     | Force the use of the user’s real name when creating messages in this area.  |
| Squish       | The current area uses the Squish message format. This is the default.   |

---

#### *Tag*

This keyword tells Maximus the name of the “area tag” for the current message area. This tag is used when writing the **Log EchoMail** file. This tag should be the same as specified for the area in your EchoMail processor’s area control file (typically **areas.bbs** or **squish.cfg**).

## 18.7. File Area Control File

### **18.7.1. Description**

The default file area control file is called **filearea.ctl**. This file defines all of the file areas accessible on a Maximus system.

### **18.7.2. Alphabetical Keyword Listing**

*ACS* <acs>

This keyword specifies the access level required to see or access this file area.

*Barricade* <menu\_name> <barricade\_file>

This keyword specifies a barricade file to be used for the current file area. As long as the current menu is <menu\_name>, the barricade privilege level from <barricade\_file> will override the user's normal privilege level. Please see section 4.6 for more information on barricade files.

*Desc* <desc>

*Description* <desc>

These keywords specify the description of the file area, as you wish it to appear on the file area menu.

*Download* <path>

This keyword tells Maximus that the files in this area can be downloaded from <path>.

*End FileArea*

This keyword tells SILT that the current file area definition is complete.

*FileArea* <area>

This keyword tells SILT to begin a file area definition. <area> is the "leaf" name of the file area; if the area is contained within a file division, the name of the division will be added to the front of the string when writing the area to the file area data file.

*FileDivisionBegin* <name> <priv> <display\_file> <description>

This keyword is used to begin a file area division. Divisions can be used to create a multi-level, hierarchical file area structure.

<name> is the name of the file area division. This should be a very short tag identifying the area type. (It should be short because it is added to the beginning of all area names declared within the division.)

<acs> is the access control string required to view the file division. Please note that the ACS specified for this division has nothing to do with the ACS required to enter a contained file area. (However, the ACS for the file areas in the division will typically be at least as restrictive as the ACS for the division.)

<display\_file> is the name of the file area display file to show when the user requests an area list of this division. This file is only used if the corresponding **Uses FileAreas** statement is enabled in the system control file. If you are not using custom file area menus, specify a "." as the filename.

**<desc>** is the description for this division, as it will appear on the file area menu.

#### *FileDivisionEnd*

This statement tells SILT to end a file area division.

#### *FileList*

This keyword specifies the location of a **files.bbs**-like list for this file area. This is useful for CD-ROM handling where there is no compatible **files.bbs** in the download directory for this area.

FB also uses this name as the “base” filename for creating compiled file information. FB removes the extension from the file you specify here, and it then adds **.dat**, **.dmp** and **.idx** extensions to store the compile file information.

#### *MenuName <orig\_menu> <replace\_menu>*

While Maximus is in this file area, when it is instructed to display the menu **<orig\_menu>**, it will instead display **<replace\_menu>**.

#### *Override <menu\_name> <option\_type> <acs> [letter]*

The **Override** keyword instructs Maximus to alter the privilege level required to access a menu option while the user is in this file area.

**<menu\_name>** is the name of the menu containing the command to be overridden. In most cases, this will be “FILE”.

**<option\_type>** is the menu option type to override. Any of the keywords from the menus control file can be used here. If you do not specify a **[letter]**, the override will apply to all options on the menu with a type of **<option\_type>**.

**<acs>** is the new ACS required to access the menu option.

**[letter]** is the optional first letter of the command to be overridden. If this letter is specified, the override will only apply to menu commands which: are on the menu called **<menu\_name>**, which have an option type of **<option\_type>** and which have a first letter of **[letter]**.

#### *Type <types>*

This keyword specifies optional flags for a file area.

**<types>** can contain zero or more of the types from Table 18.11:

Table 18.11 File Area Types

| Type       | Description   |
|------------|---|
| CD         | This area is stored on CD-ROM. This is equivalent to <b>Type Slow Staged NoNew</b> .  |
| DateAuto   | Use automatic file dating for this area. FB and Maximus will retrieve a file's size and date information from the directory specified in the <b>Download</b> path.  |
| DateList   | Use list-based file dating for this area. Both Maximus and FB parse the file dates and sizes correctly out of the files.bbs for this area without looking for the directory entry.  |
| DateManual | Use no file dating at all. The file descriptions for this area may contain size and date information, but they are not interpreted by Maximus or FB in any way.   |
| Free       | This type is equivalent to specifying <b>Type FreeTime FreeSize</b> .   |
| FreeBytes  | Both of these keywords allow all files in this area to be downloaded without adding the files to the user's file download limit. This is equivalent to placing a "/b" in the description for all files in the area.   |
| FreeSize   |   |
| FreeTime   | This keyword allows users to download all of the files in this area with no impact on their time limit. This is equivalent to placing a "/t" in the description for all files in the area.  |
| Hidden     | This area is hidden from the normal <b>File_Area</b> area listing. Users cannot see the area on the menu or change to it using the <b>File_Area</b> next/prior keys. However, the area can still be accessed by name.   |
| NoNew      | This area is on a permanent storage medium and should be excluded from new file checks.   |
| Slow       | File area is on a slow-access medium. Maximus and SILT assume that the file area always exists (and therefore they do not check the directory when doing an area list). Maximus also tries to access the directory entries in this area as little as possible.  |
| Staged     | Files from this area are copied to the <b>Stage Path</b> before a file download is initiated.<br>Use of a staging area allows file transfers to proceed at full speed, and this feature will also prevent wear and tear on a multi-disk CD-ROM changer.<br><br>When using the internal transfer protocols, Maximus copies each file to the staging area before it is transferred, deleting it immediately after the file is sent.<br><br>For external protocols, Maximus copies all of the files to the staging area at once, invokes the external protocol, and then deletes all of the files upon return. |

*Upload*

This keyword tells Maximus where to place files that are uploaded in this area.

## 18.8. Menu Control File

### 18.8.1. Description

The default menu control file is called **menus.ctl**. This file defines all of the menu options and commands that are accessible to users.

A menu definition begins with the following line:

```
Menu <name>
```

and ends with the following:

```
End Menu
```

All of the keywords between these two lines are considered part of the menu definition. A menu normally consists of a number of global menu options followed by one or more options to be displayed on the menu.

### 18.8.2. Global Menu Options

*Global* menu options may appear anywhere in a menu definition. These options can appear in any order.

*HeaderFile* <filespec> [type]

This keyword specifies the name of a custom **.bbs** file to display (or a MEX program to run) when entering a menu area.

This file is displayed when entering the menu, and it is also displayed after the user executes a command on the menu. The **HeaderFile** is always displayed before the **MenuFile**.

[type] can be zero or more of the following optional qualifiers:

- ◆ Novice
- ◆ Regular
- ◆ Expert
- ◆ RIP

If one or more **[type]** values are specified, Maximus will display the **HeaderFile** only to users who have at least one of the specified attributes. (By default, a **HeaderFile** is displayed to all users.)

For example, given the following line:

```
HeaderFile      Misc\Msghdr  RIP Regular
```

Maximus would display the **HeaderFile** to users who either have a help level of *regular* or who have *RIPscrip* graphics enabled.

A MEX program can be used for the **HeaderFile** definition by specifying a “:” as the first character in **<filespec>**. If the user is just entering the message, the argument passed to the **main** function is “1”. Otherwise, the argument passed to the **main** function is “0”.

#### *Menu <filestem>*

This keyword begins a menu definition. **<filestem>** is the root name of the menu file (without the **.mnu** extension). **<filestem>** should not include a path.

#### *MenuColor <attr>*

This command is normally only needed when using the “MenuFile <filespec>” keyword in conjunction with hotkeys. While these settings are in effect, if a user presses a key during the display of the **MenuFile**, Maximus will abort the display of the file and process the user’s input immediately.

However, the **MenuFile** may use odd color combinations, so this keyword tells Maximus to reset the color to **<attr>** before displaying the character entered by the user. **<attr>** is a numeric AVATAR color code, as described in Appendix F.

#### *MenuFile <filename> [type]*

This keyword instructs Maximus to display the specified **.bbs** file instead of generating a “canned” display for this menu. This option allows the SysOp to define a custom graphics screen instead of the standard yellow and gray menu display.

If you use this option on a message area menu, you are strongly urged to also use the **MenuLength** keyword to ensure that the menu output is displayed properly.

**[type]** can be zero or more of the following optional qualifiers:

- ◆ Novice
- ◆ Regular
- ◆ Expert
- ◆ RIP

If one or more **[type]** values are specified, Maximus will display the **MenuFile** only to users who have at least one of the specified attributes. By default, a **MenuFile** is displayed to all users.

*MenuLength <length>*

This keyword is only used in conjunction with the **MenuFile** keyword. This keyword informs Maximus that the custom **MenuFile** display file is exactly **<length>** lines long. Maximus uses this value to ensure that messages do not scroll off the screen when the custom menu is displayed.

*OptionWidth <width>*

This keyword tells Maximus to display each menu option in a space of exactly **<width>** characters. The default value for **<width>** is 20.

*Title <name>*

This keyword defines the name of the menu as it appears to the user. The value specified for **<name>** can also include external program translation characters.

### 18.8.3. Menu Option Modifiers

These modifiers are simple flags that can be placed in front of menu options. Menu option modifiers modify the operation of a menu item in some manner.

*Ctl <option>*

This modifier is obsolete.

*Conf <option>*

This modifier instructs Maximus to display the following menu option only in areas which have the **Style Conf** attribute.

*Echo <option>*

This modifier instructs Maximus to display the following menu option only in areas which have the **Style Echo** attribute.

*Local <option>*

This modifier instructs Maximus to display the following menu option only in areas which have the **Style Local** attribute.

*Matrix <option>*

This modifier instructs Maximus to display the following menu option only in areas which have the **Style Net** attribute.

*NoCLS <option>*

This modifier is only used in conjunction with the “Display\_Menu” option. The **NoCLS** modifier instructs Maximus not to clear the screen before displaying the specified menu.

*NoDsp <option>*

This modifier instructs Maximus not to display the following menu option on the generated menu display. This modifier is useful for creating hidden commands and linked menu options.

*NoRIP <option>*

This modifier instructs Maximus to display the following menu option only when the user does not have *RIPscrip* graphics enabled.

*ReRead <option>*

This modifier is only used in conjunction with the **Xtern\_Dos** and **Xtern\_Run** menu options.

This modifier instructs Maximus to re-read the user’s **lastus##.bbs** after running the external command. This option is useful when the external program modifies the user file and wishes to communicate the changes back to Maximus.

*RIP <option>*

This modifier instructs Maximus to display the following menu option only to those users who have *RIPscrip* graphics enabled.

*UsrLocal <option>*

This modifier instructs Maximus to display the following menu option only when the user is logged in at the local console. This modifier is useful for creating menu commands that can only be used locally by the SysOp.



*UsrRemote* <option>

This modifier instructs Maximus to display the following menu option only when the user is logged in from remote. This modifier is useful for hiding commands that are only useful for remote callers, such as call-back verification programs.

#### 18.8.4. Menu Option Format

Maximus supports up to 127 menu options on a single menu. The format for each menu option looks like this:

```
[modifier] <option_name> [arg] <acs> "<desc>" ["kp"]
```

[**modifier**] can be zero or more of the optional menu option modifiers, as described in the previous section.

<**option\_name**> is the name of the menu command to execute when the user selects this option. An alphabetical list of menu commands can be found in section 18.8.5.

[**arg**] is the optional argument for the menu command. This field must not be specified unless the menu command specifically requires an argument. The description for each menu command indicates whether or not the command requires an argument.

<**acs**> is the access control string required to view or execute the menu option.

<**desc**> is the description of the menu command, as it appears on the menu. The description must be enclosed in quotes. Maximus uses the first letter of the description as the selection character for the command; this means that the menu option will be executed when the user presses this character. Normally, the first letter of <**desc**> must be unique among options on a single menu.

However, Maximus also allows menu options to be connected to the function keys and arrow keys on an IBM-style keyboard. These special keys work on the local console, and they also work for any user with a terminal program that supports "DoorWay mode."

If a back-quote ("`) is specified as the first character of <**desc**>, Maximus will interpret the following number as the scan code of the key to assign to the menu option. (A list of scan codes can be found in most technical PC reference manuals. In general, scan codes are related to the placement of keys on the keyboard.)

The following key assignments are used on the default message area menu:

|       |               |                 |         |
|-------|---------------|-----------------|---------|
| NoDsp | Msg_Change    | Transient "`46" | ; alt-c |
| NoDsp | Read_Previous | Transient "`75" | ; left  |

|       |               |           |           |              |
|-------|---------------|-----------|-----------|--------------|
| NoDsp | Read_Original | Transient | "`115"    | ; ctrl-left  |
| NoDsp | Read_Next     | Transient | "`77"     | ; right      |
| NoDsp | Read_Reply    | Transient | "`116"    | ; ctrl-right |
| NoDsp | Msg_Reply     | Transient | "`16"     | ; alt-q      |
| NoDsp | Msg_Reply     | Transient | "`19"     | ; alt-r      |
| NoDsp | Msg_Kill      | Transient | "`37" "=" | ; alt-k      |

[kp] is the optional **Key\_Poke** text. This argument must be enclosed in quotes. When the user selects this menu option, Maximus will automatically insert the text in [kp] into the keyboard buffer. This functionality is identical to that provided by the **Key\_Poke** menu option.

### 18.8.5. Alphabetical Menu Option Listing

#### *Chat\_CB*

Invoke the internal multinode chat facility (in CB simulator mode). Please see section 7.2 for more information.

#### *Chat\_Page*

Prompt the user for a node number to page and then send a chat request to the specified user. After the message is sent, Maximus places the user inside the multinode chat until the other user responds to the page. Please see section 7.2 for more information.

#### *Chat\_Pvt*

Invoke the internal multinode chat (in private chat mode). Please see section 7.2 for more information.

#### *Chat\_Toggle*

Toggle the user's multinode chat availability flag.

#### *Chg\_Alias*

Change the user's alias to a new value. The new alias must not conflict with the name or alias of any other user on the system.

#### *Chg\_Archiver*

Select the default compression method for downloading QWK packets. Normally, the user is prompted to select an archiver for every download. This menu option can be used to select a default archiver and suppress the prompt.

*Chg\_City*

Change the user's city to a new value.

*Chg\_Clear*

Toggle the user's clearsreen setting.

*Chg\_Editor*

Toggle the user's full-screen editor flag. This option toggles between the line editor and the full-screen MaxEd editor.

*Chg\_FSR*

Toggle the user's full-screen reader flag. When enabled, an attractive message header is displayed to users with ANSI or AVATAR graphics. All of the fields in the message header are presented at once, and the user can move back and forth between the fields using the cursor keys or tab key.

*Chg\_Help*

Change the user's help level. Maximus supports the *novice*, *regular* and *expert* help levels.

*Chg\_Hotkeys*

Toggle the user's hotkeys setting. With hotkeys enabled, menu commands are executed as soon as the key is pressed, without waiting for the user to press *<enter>*.

*Chg\_IBM*

Toggle the user's "IBM extended ASCII" flag. This controls whether Maximus sends high-bit characters directly or whether it translates those characters to ASCII equivalents.

*Chg\_Language*

Select a new language file. Any of the language files specified in the language control file can be selected here.

*Chg\_Length*

Change the user's screen length to a new value.

*Chg\_More*

Toggle the display of “More [Y,n,=]?” prompts.

*Chg\_Nulls*

Change the number of NUL characters sent after every transmitted line.

*Chg\_Password*

Change the user’s password to a new value.

*Chg\_Phone*

Change the user’s telephone number.

*Chg\_Protocol*

Select a new default protocol. Normally, Maximus prompts the user to select a file transfer protocol for every upload and download. A default protocol can be selected to suppress this prompt.

*Chg\_Realname*

This option is obsolete.

*Chg\_RIP*

Toggle the user’s *RIPscrip* graphics setting. Enabling *RIPscrip* also forces ANSI graphics and screen clearing to be enabled.

*Chg\_Tabs*

Toggle the user’s “tab” setting. This flag tells Maximus if it can transmit tab characters to the user; if not, it will translate tabs into sequences of up to eight spaces.

*Chg\_Userlist*

Toggle the user’s “display in userlist” flag. If this option is set to *no*, the user is never displayed in the userlist. If this option is set to *yes*, the user may be displayed in the user list, depending on whether or not the **Flags Hide** attribute is specified for the user’s class in the access control file.

*Chg\_Video*

Select a new video mode. Maximus supports the TTY, ANSI and AVATAR video modes.

*Chg\_Width*

Change the user's screen width to a new value.

*Clear\_Stacked*

Clear the user's command-stack buffer. This will eliminate any leftover input in the keyboard input buffer. This option is normally linked with another menu option.

*Display\_File* <filespec>

This command displays the **.bbs** file specified by the <filespec> argument. This argument can also include external program translation characters. (However, remember that translation characters used in filenames must begin with a "+" character, not a "%" character.)

For example, to display a file called **bps<b>.bbs**, where <b> is the current baud rate, the following line can be used:

```
Display_File      D:\Bps+B          Demoted "BaudFile"
```

*Display\_Menu* <name>

This menu option instructs Maximus to display the menu specified by <name>. The <name> argument must not include a path or extension.

**Display\_Menu** calls are "flat," in that the called menu does not implicitly know how to return to the calling menu (without using an explicit **Display\_Menu** with the calling menu's name). If this behavior is not desired, see the **Link\_Menu** menu option for a more robust approach.

*Edit\_Abort*

Abort entry of the current message. This option is only valid within the line editor.

*Edit\_Continue*

Append to a message in the line editor, or return to the full-screen MaxEd display. This option is only valid within one of the editor menus.

*Edit\_Delete*

Delete a line from a message in the line editor. This option is only valid within the line editor.

*Edit\_Edit*

Modify one line within the message. This option is only valid within the line editor.

*Edit\_From*

Edit the **From:** field of a message. This option is only valid within one of the editor menus.

*Edit\_Handling*

Modify the message attributes associated with a message. This option should only be accessible to the SysOp. This option is only valid within one of the editor menus.

*Edit\_Insert*

Insert a line into the message. This option is only valid within the line editor.

*Edit\_List*

List the lines in the current message. This option is only valid within the line editor.

*Edit\_Quote*

Copy text from the message that the user is replying to and place it in the current message. This option is only valid within the line editor.

*Edit\_Save*

Save the message and place it in the message base. This option is only valid within the line editor.

*Edit\_Subj*

Edit the **Subject:** field of the message. This option is only valid within one of the editor menus.

*Edit\_To*

Edit the **To:** field of the message. This option is only valid within one of the editor menus.

*File\_Area*

Prompt the user to select a new file area.

*File\_Contents*

Display the contents of a compressed file. Maximus supports files compressed using the ARC, ARJ, PAK, ZIP, and LZH compression methods.

*File\_Download*

Download (send) a file from the file areas.

*File\_Hurl*

Move a file from one file area to another.

*File\_Kill*

Delete a file from the current file area.

*File\_Locate*

Search all file areas for a file with a certain filename or description. This command can also be used to display a list of new files.

*File\_NewFiles*

Search for new files in all file areas. This command is identical to the *[new-files]* MECCA token.

*File\_Override*

Temporarily change the download path for the current area. This function allows the SysOp to access any directory on the BBS as if it were a normal file area. See also the **File\_Raw** menu option.

*File\_Raw*

Display a raw directory of the current file area. This option shows all files in the directory, not just those contained in **files.bbs**.

*File\_Tag*

Add (“tag”) a file and place it in the downloaded queue. The files tagged by this command can be downloaded later using the **File\_Download** command.

*File\_Titles*

Display a list of all files in the current file area, including the name, timestamp and description of each file. To produce this listing, Maximus reads the **files.bbs** file for the current area.

*File\_Type*

Display the contents of an ASCII text file in the current file area.

*File\_Upload*

Upload (receive) a file.

Maximus can automatically exclude specific types of files from being uploaded. When the user uploads a file, Maximus compares the filenames of the files to be uploaded with the names specified in the **\max\badfiles.bbs** file. If an uploaded file is named in that file, Maximus will display the **\max\misc\bad\_upld.bbs** file and abort the upload.

Filenames specified in **badfiles.bbs** can be full filenames or wildcard file specifications. Spaces or newlines can be used to separate file specifications. For example:

```
MAKE$$$$.TXT MAKECASH.*
*.RBS *.GBS *.BBS
*.GIF *.JPG *.TIF
```

Uploaded files which are excluded by this list are automatically deleted and the user is not given credit for the upload.

*Goodbye*

Log off the system.

*Key\_Poke <keys>*

Insert the keystrokes specified by **<keys>** into the user’s keystroke buffer. Maximus behaves just as if the user had entered the keystrokes manually. Ensure that all spaces are replaced with underscores.

For example:



```
Key_Poke          bayl          Demoted "*List new msgs"
```

If this command is executed from the message menu, it will select the **Browse / All / Your / List** command and display a list of all new messages addressed to the current user.

External program translation characters can also be included in the **<keys>** sequence. To place an **<enter>** keystroke in the input sequence, a **“;”** or **“|”** character will work with most (but not all) commands.

Title: Af  
Creator:

Keys can also be implicitly placed in the keyboard buffer by placing an extra set of quotation marks after the name of a menu option. For example, the following menu option:

```
Msg_Browse          Demoted "*List new msgs" "ayl"
```

automatically places the **“a,” “y”** and **“l”** characters in the keyboard buffer before executing the **Browse** command.

#### *Leave\_Comment*

Place the user in the message editor and address a message to the SysOp. The message is placed in the area defined by **Comment Area** in the system control file.

#### *Link\_Menu <name>*

This menu option can be used to nest menus. When menu **<name>** is displayed by this option, the **Return** menu option can be used to return back to the calling menu. Maximus supports up to eight nested **Link\_Menu** options.

For example, if the following option is placed on your main menu:

```
Link_Menu CHAT          Demoted      "/Chat menu"
```

then placing the following option on your chat menu allows Maximus to return to the main menu when the **“M”** key is selected:

```
Return          Demoted      "Main menu"
```

#### *Msg\_Area*

Prompt the user to select a new message area.

*Msg\_Browse*

Invoke the **Browse** function. This option allows the user to selectively read, list, or pack messages for the QWK mail packer. Messages can be selected by area, by message type, and by a user-defined search of the message header and body.

*Msg\_Change*

Modify an existing message. Maximus only allows users to modify messages that have *not* been:

- ◆ read by the addressee,
- ◆ scanned out as EchoMail, or
- ◆ sent as NetMail

However, if the user's class has the **MailFlags MsgAttrAny** attribute specified, Maximus will allow the user to modify the message anyway.

*Msg\_Checkmail*

Invoke the built-in mailchecker. This option is equivalent to the *[msg\_checkmail]* MECCA token.

*Msg\_Current*

Redisplay the current message.

*Msg\_Download\_Attach*

Lists unreceived file attaches addressed to the current user. This command then prompts the user to download each file attach.

*Msg\_Edit\_User*

Invoke the user editor and automatically perform a search on the user listed in the **From:** field of the current message. This option is normally only accessible to the SysOp.

*Msg\_Enter*

Create a new message in the current area.

*Msg\_Forward*

Forward a copy of the current message to another user.

*Msg\_Hurl*

Move a message from one area to another.

*Msg\_Kill*

Delete a message from the current area. The message must be either **To:** or **From:** the current user. (Maximus will also allow users in a class with the **MailFlags ShowPvt** attribute to delete messages which are addressed to other users.)

*Msg\_Kludges*

Toggle display of the <ctrl-a> “kludge” lines at the beginning of messages. These lines normally hold message routing and control information. This option is normally only available to the SysOp.

*Msg\_Reply*

Reply to the current message. The reply is placed in the current area.

*Msg\_Reply\_Area <area>*

Reply to the current message in another area.

If the <area> argument specifies an explicit message area name, the reply is automatically placed in the indicated area. If a “.” is specified, the user is prompted to select a message area for the reply.

*Msg\_Restrict*

Limit QWK message downloads on the basis of message date. Users can set this field so that messages that arrived on the system prior to a certain date are not downloaded. This date remains in effect for the current session only.

*Msg\_Tag*

Tag (select) specific message areas of interest. The list of tagged areas can be later recalled when using the **Msg\_Browse** command.

*Msg\_Unreceive*

“Unreceive” the current message. This option resets the “received” flag for the current message, regardless of the message’s author. This option should normally only be available to the SysOp.

*Msg\_Upload*

Create a message by uploading a file, rather than invoking one of the internal editors. Maximus still prompts the user for the message header information, but after the header is created, it prompts the user to upload an ASCII text file.

*Msg\_Upload\_QWK*

Upload a **.rep** reply packet generated by a QWK off-line reader.

*Msg\_Xport*

Export a message to an ASCII text file. This option allows the user to specify an explicit path and filename, so only the SysOp should be able to access this command.

To print a message, select the **Msg\_Xport** option and specify a filename of “prn”.

*Press\_Enter*

Set the text color to white and prompt the user to press <enter>. This option is normally linked with another menu option.

*Read\_DiskFile*

Import an ASCII text file from the local disk and place it in the current message. This option can only be used from one of the editor menus. This option allows the user to specify an explicit path and filename, so it should only be available to the SysOp.

*Read\_Individual*

Read a specific message number in the current area.

*Read\_Next*

Read the next message in the current area.

*Read\_Nonstop*

Display all of the messages in the area without pausing after each message. If the last reading command was **Read\_Next**, messages are displayed in forward order. Otherwise, if the last reading command was **Read\_Previous**, messages are displayed in reverse order.

*Read\_Original*

Read the original message in the current thread. See also **Read\_Reply**.

*Read\_Previous*

Read the previous message in the current area.

*Read\_Reply*

Display the next message in the current thread. See also **Read\_Original**.

*Return*

Return from a menu that was called with the **Link\_Menu** option.

*Same\_Direction*

Continue reading messages in the same direction, as specified by the last **Read\_Previous** or **Read\_Next** menu option.

*User\_Editor*

Invoke the system user editor. This option should only be accessible to the SysOp.

*Userlist*

Display a list of all users on the system. Users who disable the “In UserList” setting in their user profile are not displayed. In addition, users in classes with the **Flags Hide** attribute are never displayed.

*Version*

Display the Maximus version number and credit information. Maximus prompts the user to press <enter> after displaying this screen.

*Who\_Is\_On*

On a multinode system, this displays the names, task numbers, and status of users who are logged on.

*Xtern\_Dos* <cmd>  
*Xtern\_Erlvl* <errorlevel>[\_<cmd>]  
*Xtern\_Run* <cmd>

Run the external program specified by <cmd>. If <cmd> has arguments, ensure that all spaces are replaced with underscores.

Please see section 6 for more information on running external programs.

*Yell*

Page the system operator. Maximus will play one of the yell tunes if yelling is currently enabled.

## 18.9. Access Control File

### 18.9.1. Description

The default access control file is **access.ctl**. This file defines the user classes and privilege levels for all users on the system.

### 18.9.2. Alphabetical Keyword Listing

*Access* <name>

This keyword begins a class definition. <name> is the unique symbolic name for this class. This name may be used interchangeably with the argument specified for **Level**.

*Calls* <no>

This keyword specifies the maximum number of times per day that users of this class are allowed to log on. If <no> is -1, users can log on an unlimited number of times.

*Cume* <mins>

This keyword specifies the maximum amount of time per day that users of this class are allowed to log on the system. See also the **Time** keyword.

*Desc <desc>*

This keyword specifies an optional description for the privilege level. If no description is specified, the name of the class itself is used.

*End Access*

This keyword marks the end of an access level definition.

*FileLimit <kbs>*

This keyword specifies the maximum number of kilobytes that users in this class are allowed to download per day.

*FileRatio <amt>*

This keyword specifies the download:upload ratio for users in this class. After exceeding the **RatioFree** level of downloads, the user must upload files such that the download:upload ratio is at least **<amt>**. Otherwise, Maximus will not allow the user to download files.

*Flags <words>*

Set a number of attributes for users of this class.

Zero or more of the options from Table 18.12 can be specified for **<words>**:

**Table 18.12 Access Flags**

| Word         | Description  |
|--------------|--|
| DloadHidden  | Users can download files which are hidden or not listed in the files list for the current file area.   |
| Hangup       | Users of this class are not allowed to log on. Maximus will hang up immediately when a user of this class calls the system.                  |
| Hide         | Users of this class are not displayed in the system user list.   |
| NoFileLimit  | The download byte/ratio limits do not apply to users in this class.  |
| NoTimeLimit  | Disables time limit and input timeout checking for users in this class.  |
| NoLimits     | This is equivalent to specifying both <b>NoFileLimit</b> and <b>NoTimeLimit</b> .  |
| ShowAllFiles | When displaying a file list, display all files, even those which were hidden by placing an “@” as the first character in the file area list. |
| ShowHidden   | Users in this class can see all users in the user list, regardless of “do not display in list” settings.                                     |

---

|           |  |
|-----------|--|
| UploadAny | Users in this class can upload any type of file, bypassing checks for <b>.bbs</b> , <b>.gbs</b> , and <b>.rbs</b> files, and also bypassing the checks for files listed in <b>badfiles.bbs</b> . |
|-----------|--|

---

*Key* <letter>

This keyword defines the one-character key used to specify this user class level. This key is used by some of the obsolete MECCA tokens (including [*?below*], [*?above*], [*?line*], and [*?file*]).

*Level* <lvl>

This keyword specifies the numeric privilege level for this user class. <lvl> must be between 0 and 65535 (inclusive). <lvl> must be unique among all user classes.

*LoginFile* <filename>

This keyword specifies the file to be displayed to users in this class as soon as they log on.

*LogonBaud* <baud>

Users of this class must have a speed of at least <baud> before they are allowed to log on.

*MailFlags* <words>

These flags control a number of options related to entering messages and viewing mail.

Any of the values from Table 18.13 can be specified for <words>:

**Table 18.13 Access Mail Flags**

| Type        | Description   |
|-------------|---|
| Editor      | If an external message editor is defined, both local and remote users in this class are permitted to use it.                    |
| LocalEditor | If an external message editor is defined, local users in this class are permitted to use it.                                    |
| MsgAttrAny  | Users in this class are allowed to modify any message or set any attribute in a NetMail message.                                |
| NetFree     | Users in this class are not charged for entering NetMail messages. The user's NetMail credit and debit fields are not modified. |
| NoRealName  | Users in this class will never have the REALNAME kludge added to messages that they create.                                     |



|             |  |
|-------------|--|
| ShowPvt     | Users in this class can see all messages, regardless of the message addressee or private flag. |
| WriteRdOnly | Users in this class can post messages in areas marked as <b>Style ReadOnly</b> .               |

---

*OldPriv <value>*

This keyword specifies the “Maximus 2.0 compatibility privilege level”. Maximus 3.0 itself does not use this value; however, SILT will use this number when writing data files that are compatible with Maximus 2.0.

This field is not optional. (If you create additional user classes, copy the **Old-priv** value from one of the existing classes with a similar privilege level.)

*RatioFree <kbs>*

Users in this class are allowed to download **<kbs>** kilobytes of files before the **FileRatio** limit is applied.

*Time <mins>*

Users in this class are allowed to log on for up to **<mins>** minutes per session.

*UploadReward <value>*

This keyword tells Maximus how much time to give back to users who upload files.

A **<value>** of 100% adds back only the amount of time that the user spent uploading the file, so the user will have the same amount of time left when the upload is complete.

To give users extra time for uploading files, use a **<value>** in excess of 100%.

To provide no compensation for uploads, set the reward to 0%.

*UserFlags <value>*

This keyword specifies an optional set of flags. These class flags can be used in MEX programs to test for various class attributes. Maximus itself does not use this field.

This value may be given in decimal or hexadecimal. (Hexadecimal constants are specified using the “0x” or “\$” prefix.)

*XferBaud* <baud>

Users in this class must have a speed of at least <bau**d**> to download files.

## 18.10. Protocol Control File

The default protocol control file is **protocol.ctl**. Maximus can directly use external protocols such as DSZ, MPt, Kermit, and others. Maximus has a configurable, control-file-driven protocol system which can interface to almost any type of external protocol.

In addition to “standard” protocols such as DSZ, Maximus also supports “Opus-compatible” protocols, such as OKermit, OASCII and others. These protocols must also be defined in the protocol control file, but they use a slightly different declaration format.

### 18.10.1. Alphabetical Keyword Listing

*ControlFile* <filespec>

This keyword defines the name of the control file to create for this protocol. Maximus will write text to this control file indicating the actions that the protocol is to perform (such as “send file X” or “receive files to directory Y”).

The exact text written to this file is controlled by the **DownloadString**, **UploadString** and **Type Opus** keywords.

If you have a multinode system, ensure that the task number is included in the name of the control file (using the %K token). Otherwise, the control file could get overwritten by another task.

*DescriptWord* <num>

When parsing the upload log created by the protocol, this keyword defines the “word number” of the description for the uploaded file.

For each line in the upload log, Maximus will split the line into a number of words. Maximus will search for the <num>th word after it finds the **Upload-Keyword** string. Everything from that word until the end of the string is considered the description for the uploaded file. If the upload log does not include descriptions, you must specify a value of 0 for <num>.

For example, if the upload log looks like this:

```
= 10 Sep 14:10:10 FROG Got \upl\docs.zip Maximus docs
```

**DescriptWord** should have a value of 2, since the “Maximus docs” description begins two words after the **UploadKeyword** of “Got.”

#### *DownloadCmd <cmd>*

This keyword specifies the command to execute when a user requests a download using the current protocol.

For an Opus-compatible protocol, the following format must be used:

```
DownloadCmd <n>.Exe <n>%K.Ctl -p%p -b%b -t%k -m%d -f%D -r%t
```

where **<n>** is the name of the external protocol, such as “ASCII” or “Kermit.”

#### *DownloadKeyword <keyword>*

When parsing the download log created by an external protocol, Maximus uses this keyword to identify the log entries that indicate that the user has downloaded a file.

For example, if the protocol writes out lines in this format when the user downloads a file (as do Opus-compatible protocols):

```
Sent c:\path\filename.zip
```

you should specify a **DownloadKeyword** of “Sent.” To search for a string containing spaces, enclose **<keyword>** in quotes.

#### *DownloadString <cmd>*

Maximus will place this string in the protocol control file when it wishes to send a file to the user.

This line is written to the control file once for each file selected by the user. If the “%s” token is included in the command string, it is replaced with the name of the file to be sent.

For example, with the following definition:

```
DownloadString Send %s
```

Maximus will create a protocol control file that contains entries of the form:

```
Sent d:\file\maxutil\uedit.zip
Sent d:\file\netutil\nodelist.zip
```

For Opus-compatible protocols, use a **DownloadString** of “Send %s”.

*End Protocol*

This keyword marks the end of a protocol definition.

*FilenameWord <num>*

When parsing the upload log created by the protocol, this keyword defines the “word number” of the name of the uploaded file.

For each line in the upload log, Maximus will split the line into a number of words. Maximus will search for the <num>th word after it finds the **UploadKeyword** string. That word is the name of the uploaded file.

For example, if the upload log looks like this:

```
= 10 Sep 14:10:10 FROG Got \upl\docs.zip Maximus docs
```

**FilenameWord** should have a value of 1, since the **\upl\docs.zip** filename begins one word after the **UploadKeyword** of “Got.”

*LogFile <filespec>*

This keyword defines the name of the log file created by the external protocol. The “%K” external program translation character can be used to embed the task number in the filename.

When the protocol returns, Maximus scans this file for download and upload file information, as specified by the **DownloadString** and **UploadString** keywords.

*Protocol <name>*

This keyword identifies the beginning of a protocol definition. The <name> parameter is used to identify the name of the protocol on the protocol menu. Consequently, the first character of <name> must be unique.

*Type Batch**Type Bi**Type Opus*

These modifiers are used to set certain flags for the external protocol.

Zero or more of the optional modifiers from Table 18.14 can be included:

**Table 18.14 Protocol Types**

| Keyword | Description  |
|---------|--|
| Batch   | The specified protocol accepts more than one file at a time. The user is not prompted to enter the names of files to upload.             |
| Bi      | The protocol can both send and receive files at the same time. Maximus scans the protocol log file for both upload and download entries. |
| Opus    | Maximus should generate Opus-compatible information at the beginning of the protocol control file.                                       |

*UploadCmd <cmd>*

This keyword specifies the command to execute when a user requests an upload using the current protocol.

For an Opus-compatible protocol, the following format must be used:

```
UploadCmd <n>.Exe <n>%K.Ctl -p%p -b%b -t%k -m%d -f%D -r%t
```

where **<n>** is the name of the external protocol, such as “ASCII” or “Kermit.”

*UploadString <cmd>*

This keyword defines the string that Maximus places in the protocol control file to request that a file be uploaded.

For non-batch protocols, a “%s” in **<cmd>** translates into the path and filename of the file to be received.

For batch protocols, a “%s” in **<cmd>** translates into the upload directory.

For Opus-compatible protocols, **<cmd>** should be “Get”.

*UploadKeyword <keyword>*

When parsing the upload log created by an external protocol, Maximus uses this keyword to identify the log entries that indicate that the user has uploaded a file.

For example, if the protocol writes out lines in this format when the user uploads a file (as do Opus-compatible protocols):

```
Got c:\path\filename.zip
```

you should specify a **UploadKeyword** of “Got.” To search for a string containing spaces, enclose **<keyword>** in quotes.

### 18.10.2. Examples

Sample protocol entries for BiModem, DSZ (Zmodem MobyTurbo), OASCII, OKermit and MPt are contained in the distribution version of the **protocol.ctl** file. However, these protocol entries are commented out by default. To enable a protocol, simply uncomment all of the lines belonging to that protocol.

If you are using an Opus-compatible external protocol, the entry in **protocol.ctl** should have the following format. (Replace each instance of <name> with the name of the external protocol.)

```
Protocol <name>
  Type Batch
  Type Opus
  LogFile <name>%K.Log
  ControlFile <name>%K.Ctl
  DownloadCmd <name>.Exe <name>%K.Ctl -p%p -b%b -t%k -m%d -f%D -%t
  UploadCmd <name>.Exe <name>%K.Ctl -p%p -b%b -t%k -m%d -f%D -r%t
  DownloadString Send %s
  UploadString Get %s
  DownloadKeyword Sent
  UploadKeyword Got
  FilenameWord 1
  DescriptWord 4
End Protocol
```

## 18.11. Event File

Maximus 3.0 includes an internal event file manager. When using Maximus in WFC mode, the event manager can be used to run external programs at predetermined times. The event manager also controls when the **Yell** command can be used to page the SysOp.

All events are defined in an ASCII file named **events##.bbs**, where **##** is the Maximus node number (in hexadecimal). Maximus will automatically compile the ASCII-based event file into a binary **events##.dat** file when it starts up.

By default, Maximus will always load the event file for the node specified by the **Task** keyword in the control file, or for the node specified using **-n** on the command line. However, one event file can be used for an entire multinode system as long as the event file contains only yell events. To override the **“##”** used in **events##.bbs**, use the **-e** command line parameter.

Every line in the event file has the following format:

```
Event <day> <start> [end] [flags...]
```

**<day>** specifies the day on which this event is to be executed. Valid days are “Sun,” “Mon,” “Tue,” “Wed,” “Thu,” “Fri,” and “Sat.” To run an event on every day of the week, specify “All.” To run an event only on weekdays, specify “WkDay.” To

run an event only on weekends, specify “WkEnd.” To run an event on a combination of days, separate each day with a “|”. (For example, “Sun|Mon|Tue” specifies an event that runs on Sunday, Monday and Tuesday.)

<start> is the starting time for the event in 24-hour format.

[end] is the optional ending time for the event, also in 24-hour format. External events do not require an ending time, but yell events do.

Following the starting and ending time are zero or more flags. Any or all of the flags from Table 18.15 can be specified:

**Table 18.15 Event Flags**

| Flag          | Description   |
|---------------|---|
| exit=<erl>    | This tells Maximus to exit with an errorlevel of <erl> as soon as the event starts. This flag is valid only when using WFC mode.  |
| bells=<num>   | This specifies the number of bells (or tunes) that Maximus plays when a user yells during this event. This flag activates the yell function for the specified time period.  |
| maxyell=<num> | This specifies the maximum number of times that a user can yell (without the SysOp answering) during one session.   |
| tune=<name>   | This specifies the tune to play during the current event. <name> can be any single word up to 32 characters long. Maximus will search the tunes.bbs file to find the tune with the specified name.<br><br>If no tune is specified, Maximus will make simple beeping noises. To have Maximus select a tune at random, use “tune=random”. |

For example, given the following event line:

```
Event Wed|Thu|Fri 20:00 23:59 exit=9 bells=3 maxyell=2 tune=StarTrk
```

Maximus exits with errorlevel 9 at the beginning of the event. If a user yells, Maximus plays the “StarTrk” tune (from **tunes.bbs**) up to 3 times for each yell. The user would be allowed to yell only twice during one session.

## 18.12. Language Translation File Reference

If you wish to change the language source in **english.mad**, you should pay attention to these points:

- Ensure that you do not change the order of the statements within the file. Disaster will result if the strings get out of order. You can add and delete blank lines or comments, but leave the order of the strings alone.
- After changing the language file source, the file must be recompiled with MAID to create the **.lrf** version of the language.
- Finally, if you wish to create a modified language for other people to use, you can simply copy **english.\*** to **mylang.\*** (or whatever the new language is to be called). You can then add a **Language Mylang** statement in the language control file.

The exact definitions for the strings in the **english.mad** are version-dependent, so they are not described here. Many of the strings can be changed by simply replacing the text in the string with the appropriate translation.

However, for strings that include formatting characters (such as “%s,” “%c” and others), ensure that the order of these characters is not modified. The text between the groups of formatting characters can be changed, as long as the formatting characters remain in the same order relative to each other.

In addition, some of the strings in the language file have an implied maximum length. There is no easy way to determine the maximum length of a language file string. However, if you expand one of the language strings by a significant amount and your copy of Maximus no longer works when using that language file, try reducing the length of the string.

Finally, you can use the *user heap* concept to add language file support for MEX programs. User heaps are user-definable language string heaps, similar to the heaps in the system language file, except that they are used exclusively by MEX programs.

A user heap is defined in the language file by using a header of the form:

```
=heapname
```

(This is used instead of the regular heap header of “:heapname”).)

MAID automatically exports user heaps to the **language.mh** file when it is run. For every language string defined in a user heap, MAID generates a “str\_<name>” macro that can be used from within a MEX program to reference the string.

To use a user heap from a MEX program, you must:

1. create a **<heapname>.lh** file, containing the “=<heapname>” header and any strings that you want to place in the heap,



2. edit `\max\lang\user.lh` and add an “#include” directive to include the `.lh` file from step 1,
3. add a “#define INCL\_<heapname>” directive at the top of your MEX program,
4. add a “#include <language.mh>” directive at the top of your MEX program, just below the #define from step 3, and
5. call the function called `init_lang_<heapname>` from within your MEX program’s `main` function.

Having done this, any of the strings declared in the `<heapname>.lh` file can be accessed from your MEX program by appending the “str\_” prefix to the beginning of the string name.



MEX programs that use a user heap must be recompiled if the portion of the language file containing that heap is modified.

For an example of user heaps, please see `\max\m\mexchat.mex` and the associated `\max\m\mexchat.lh` file.



---

# Appendices

## Appendix A: Common Problems

This section describes some of the common problems that some Maximus SysOps experience:

---

**PROBLEM** Users cannot upload QWK messages. Maximus reports “invalid message area” whenever a user tries to upload a message.

**SOLUTION** Maximus needs some way to determine whether or not a user is allowed to upload a message to an area. To do this, it looks for the menu named “MESSAGE” and tries to find a **Msg\_Upload** command on it. Maximus uses the privilege level for this command to determine whether or not the user can access the command.

However, if you have renamed your message menu, Maximus will not know where to look and it will not allow users to upload messages. To solve the problem, you can either:

- add a **MenuName** keyword that specifies the correct message menu name for all of your message areas, or
- create a menu called “MESSAGE” and add a **Msg\_Upload** option to it. This menu does not need to be referenced by any other menus; just as long as the menu is called “MESSAGE,” Maximus will be able to read the menu and permit QWK uploads.

---

**PROBLEM** Maximus is not adding an origin line or a tear line to messages originating from my system, but this only seems to be happening in certain areas.

**SOLUTION** You probably specified the wrong area type in your message area control file. Ensure that you have included a **Style Echo** in the message area definition.

---

**PROBLEM** When I run an external program, Maximus tries to access one of my CD-ROMs.

**SOLUTION** You forgot to edit the **Save Directories** definition in the system control file. Before running an external program, Maximus tries to save the current directory for all drives indicated in that statement. If you accidentally specify the drive letter for a floppy or CD-ROM drive, Maximus will try to access the disk when it shells out.

---

**PROBLEM**    When I try to view a file containing ANSI graphics, it comes out garbled and I can see only the text version of the ANSI commands.

**SOLUTION**   Do not use ANSI graphics directly. Use the supplied ANS2BBS utility to convert your ANSI screens into a Maximus **.bbs** file.

---

**PROBLEM**    When using the AVATAR graphics mode, users report that everything has changed colors, the full-screen editor does not work, and other display-related problems.

**SOLUTION**   Your user is probably using Telix for DOS. Early versions of Telix have bugs in the AVATAR emulation code.

---

**PROBLEM**    When a user presses <ctrl-c>, Maximus stops sending output to the user, even though everything still looks fine on the local console.

**SOLUTION**   Turn off the **Send Break to Clear Buffer** feature in the system control file. Many modems do not support this feature.

---

**PROBLEM**    Callers sometimes do not see the end of the **\max\misc\byebye.bbs** file. They report that Maximus hangs up before it finishes displaying the file.

**SOLUTION**   Place several *[pause]* MECCA tokens at the end of **\max\misc\byebye.bbs**. Modems which have transmit buffers make no effort to empty the buffer before hanging up on remote callers; once Maximus sends the file to the modem, it assumes that the file has also been received by the caller, so it hangs up right away. Adding the pauses gives the modem enough time to display the text to the user.

## Appendix B: Error Messages

This section describes some of the common error messages that you may see in the system log file.

*ANSI sequence found, area XX msg YY*

This warning is generated by the Maximus security system. This warning indicates that it found ANSI codes embedded in the header of a particular message.

*Barricade file priv, 'XXX'?*

Maximus could not make any sense out of an ACS specified in a barricade file.

*Can't find 'XXX'*

Maximus expected to find a file in a certain location, but it was not able to find it in the right place.

*Can't find barricade file XXX*

Maximus could not find the file specified in a **Barricade** statement for a message or file area.

*Can't find class record*

Maximus was unable to find the class record (in the access control file) for a particular user's privilege level. Check the user's privilege level and ensure that it corresponds to a class entry in the access control file.

*Can't open 'XXX'*

*Can't read 'XXX'*

*Can't write 'XXX'*

These messages indicate that Maximus was looking for a particular file, but it was unable to open/read/write it. These messages could also indicate some sort of "disk full" condition.

*Err: Lastread ptr xlinkd, usr#nnn*

A user's last-read pointer has become crosslinked. This usually indicates that an external utility has damaged your user file. To fix this problem, run "cvtusr -l".

*Invalid UL path, area X*

The upload path specified for area X does not exist.

*Invalid current pwd 'XXX'*

The user tried to change his or her password in the Change Setup section, but the user failed to correctly enter the current password.

*Invalid custom cmd: 'X'*

There is an invalid character in a one of the **Format XxxFormat** definitions in the system control file. Fix the sequence and recompile.

*Invalid outside cmd: 'X'*

You attempted to use an invalid character as an external program translation character. Such sequences are normally used for external programs or when writing to a file with the *[write]* MECCA token.

*Invalid outside errorlevel*

This means that you specified an invalid errorlevel for an errorlevel exit. Valid errorlevels are 5 through 254 inclusive.

*MEM:ndir**MEM:nmsga**MEM:nmsgb*

These messages are displayed when Maximus runs out of memory. In the DOS version of Maximus, give it more conventional memory.

*Max nest lim. exceeded, XXX aborted*

This message is displayed when you have tried to *[link]* a **.bbs** file more than 8 levels deep.

*No mem for delete buf**No mem for lastread scan**Not enough mem*

These mean that Maximus is short on memory. See *MSG:ndir*.

### *Null ptr/XXX*

A critical error occurred in the Maximus code. Please report this error to Lanius Corporation, along with the circumstances under which the *Null Ptr* message was generated.

### *OA-MEMOVFL*

See *MEM:ndir*.

### *Unknown option type 'XXX'*

Maximus found an invalid menu option number in a menu file. Try recompiling the menu file.

### *Upload 'ABC.BBS' renamed to 'ABC.BBX'*

This means that a user tried to upload a file with an extension of **.bbs**. Only users in a class that has the **Flags UploadAny** attribute are allowed to upload files with an extension of **.bbs**.

### *User gave device/path 'XXX'*

### *User supplied path 'XXX'*

These messages are generated by the Maximus security system when a user specifies an explicit path or device.

For example, if the user tries to specify an upload filename called **c:\max\virus.com**, Maximus generates a log entry of “User supplied path 'c:\max'.” Maximus will automatically strip off the path, but this message indicates that a user may be trying to do devious things.

## Appendix C : Command Line Switches

This section describes the format of the Maximus command line. Maximus can be invoked as shown below:

```
max [prm_name] [switches ...]
```

[**prm\_name**] specifies the optional name of the Maximus parameter file to use. By default, Maximus will read the system information from **max.prm**.

[**switches**] can be zero or more of the following optional command line switches. If no switches are specified, Maximus starts in local mode.

Table C.1 lists the command line switches supported by Maximus:

**Table C.1 Maximus Command Line Switches**

| Switch | Description  |
|--------|--|
| -b<x>  | If used in conjunction with the -w switch, this specifies the maximum system baud rate. Otherwise, this switch informs Maximus of the speed of the incoming caller, as passed on by the program that answered the phone. See also the -s switch for information on baud rates and high-speed modems. |
| -c     | Create a user.bbs file. This command is normally only needed the first time that Maximus is run. Maximus automatically grants SysOp privileges to a user who logs on when the -c switch is used.   |
| -e<x>  | Use <x> as the decimal “task number” for reading event files. Please see section 18.11 for more information.   |
| -j<x>  | “Jam” keystrokes into the keyboard buffer. This option is useful for automatically logging on as a user. To imbed spaces in the jam command, you must enclose the entire parameter in double quotes. For example, to automatically log on as “Joe SysOp”:  |

```
max -k "-jJoe SysOp;iPd"
```

The “-j-” modifier can be used to completely clear the keyboard buffer. This forces Maximus to display the **logo.bbs** file, even for local logons.

|       |  |
|-------|--|
| -k    | Log on in local mode (default).  |
| -l<x> | Write the system log to <x>, instead of the filename specified in the system control file. If <x> is blank, no log file is used. |
| -m<x> | Override the Multitasker definition in the system control file. The following are acceptable values for <x>:<br>d — DoubleDOS    |



|       |   |
|-------|---|
|       | q — DESQview  |
|       | m — PC-MOS  |
|       | w — Windows 3.1 or Windows 95   |
|       | n — No multitasker  |
| -n<x> | Select the node number for this task. This overrides the Task definition in the system control file.  |
| -p<x> | Selects the COM port number (or port handle for OS/2) for the current session.  |
| -r    | Restart a session that was previously ended using a Xtern_Erlvl exit. Please see section 6 for more information.  |
| -s<x> | Use <x> as the locked baud rate. Maximus will always communicate with the COM port at the rate specified here. However, it will continue to calculate file transfer times using the value specified for -b. |
| -t<x> | Do not allow the current user to remain on-line for longer than <x> minutes. This command allows a front end mailer to ensure that a user does not overrun an internal mailer event.                        |
| -u    | Automatically run the system user editor without logging in. The -u switch runs the user editor in normal mode; the -uq switch runs the user editor with hotkeys enabled.                                   |
| -uq   |   |
| -vb   | (DOS only.) Selects the Maximus video mode. -vb enables the BIOS video mode; -vi enables the IBM video mode.  |
| -vi   |   |
| -w    | Run in Waiting for Caller mode. Please see section 5.1 for more information.  |
| -xc   | Disable carrier drop detection. Maximus will not monitor the DCD line to determine if a user has dropped carrier.   |
| -xd   | Disable automatic DTR dropping when Maximus ends. When a user logs off, Maximus simply sends the Busy string without changing DTR.  |
| -xj   | Disable the local <alt-j> shell feature from within Maximus. (However, this does not disable the <alt-j> sequence from within WFC mode.)  |
| -xt   | Disable Maximus's internal trap logging feature. Maximus will pause and display an error pop-up instead of just logging the crash to the log file (OS/2 only).  |
| -xz   | Disable the internal Zmodem protocol.   |

---

## Appendix D: Local Keystrokes

Table D.1 describes the keystrokes that can be used on the SysOp console while a user is on-line.

**Table D.1 Local Keystrokes**

| Key      | Description   |
|----------|---|
| <esc>    | Abort the current SysOp operation. This key will dismiss a pop-up window, abort a file transfer, turn off keyboard mode, or exit chat.  |
| <space>  | Displays the user's statistics in a pop-up window.  |
| A        | Enable local keyboard mode.   |
| L        | Lock the user's privilege level. The user's privilege level will be restored when the user logs off or when you press "U."  |
| N        | Enable Snoop Mode. Maximus will display output on the local screen.   |
| O        | Disable Snoop Mode.   |
| S        | Set privilege level. This displays a pop-up window that can be used to adjust the user's privilege level and keys.  |
| U        | Restore a user's privilege level after a prior lock operation.  |
| Z        | Zero the user's cumulative on-line time for today. This is useful if the user almost overran the <b>Cume</b> limit for the user class, but if you still want to allow the user to call back again later in the day.   |
| 1..8     | Toggle the specified key number. These toggles only work for keys 1 through 8. See the "S" keystroke to toggle other keys.  |
| +        | Promote the user's privilege level to a higher user class.  |
| -        | Demote the user's privilege level to a lower user class.  |
| !        | Toggles the noise made by the Yell command.   |
| =        | Display the current user's password. This feature does not work for users with encrypted passwords.   |
| ?        | Display the user's statistics (as with <space>) and turn off Snoop.   |
| <up>     | Add one minute to the user's time.  |
| <pgup>   | Add five minutes to the user's time.  |
| <down>   | Subtract one minute from the user's time.   |
| <pgdn>   | Subtract five minutes from the user's time.   |
| <alt-c>  | Initiate chat mode with the current user. Use the <esc> key to exit chat mode.  |
| <alt-j>  | Shell to the operating system.  |
| <alt-n>  | Toggle the "Nerd" setting for the current user. When the nerd flag is enabled, the user's yells do not make any noise on the local console.   |
| <alt-d>  | Generate fake line noise and drop carrier on the user.  |
| <ctrl-x> | Immediately disconnect the current user.  |
| <Fx>     | Pressing a function key while a user is on-line will display one of the \max\misc\<f>*.bbs files. For example, pressing <f1> displays the f1.bbs file. Similarly, <ctrl-fX>, <shift-fX> and <alt-fX> will display the related cf*.bbs, sf*.bbs and af*.bbs files. |

## Appendix E: User Editor Keystrokes

In addition to the highlighted command letters that appear on the user editor screen, Table E.1 lists a number of other keystrokes supported by the user editor:

**Table E.1 User Editor Keystrokes**

| Key | Description   |
|-----|---|
| "   | Undo changes. This undoes all changes made to the current user record.  |
| '   | Find the next user. This continues a search started with the "~" key.   |
| +   | Display the next user.  |
| -   | Display the previous user.  |
| /   | Redraw the screen.  |
| =   | Toggle the display of the user's password. (This option only works for users with unencrypted passwords.)   |
| ?   | Display help on user editor commands.   |
|     | Purge users. This function deletes all users who have the "deleted" flag set in their user record.  |
| ~   | Find a user. Maximus prompts you for the name of the user to find (or a part thereof). Maximus will also search in the alias and phone number fields. |
| C   | Create a user. A new user record is appended to the end of the user file.   |
| D   | Toggles the "deleted" flag for the current user. The user is not actually removed from the user file until you do a purge (" ").                      |
| J   | Jump to the last user in the user file.   |

## Appendix F: AVATAR Colors

This section lists all of the AVATAR color codes that can be used in **.bbs** files.

To use this chart, first look in the left-hand column to find the row with the required foreground color. Next, look across the top of the chart to find the column with the required background color. The color number is at the intersection of the foreground row and the background column.

| Background color | Intensity | Black | Blue | Green | Cyan | Red | Magenta | Yellow | White |
|------------------|-----------|-------|------|-------|------|-----|---------|--------|-------|
| Black            | low       | 0     | 16   | 32    | 48   | 64  | 80      | 96     | 112   |
|                  | high      | 8     | 24   | 40    | 56   | 72  | 88      | 104    | 120   |
| Blue             | low       | 1     | 17   | 33    | 49   | 65  | 81      | 97     | 113   |
|                  | high      | 9     | 25   | 41    | 57   | 73  | 89      | 105    | 121   |
| Green            | low       | 2     | 18   | 34    | 50   | 66  | 82      | 98     | 114   |
|                  | high      | 10    | 26   | 42    | 58   | 74  | 90      | 106    | 122   |
| Cyan             | low       | 3     | 19   | 35    | 51   | 67  | 83      | 99     | 115   |
|                  | high      | 11    | 27   | 43    | 59   | 75  | 91      | 107    | 123   |
| Red              | low       | 4     | 20   | 36    | 52   | 68  | 84      | 100    | 116   |
|                  | high      | 12    | 28   | 44    | 60   | 76  | 92      | 108    | 124   |
| Magenta          | low       | 5     | 21   | 37    | 53   | 69  | 85      | 101    | 117   |
|                  | high      | 13    | 29   | 45    | 61   | 77  | 93      | 109    | 125   |
| Yellow           | low       | 6     | 22   | 38    | 54   | 70  | 86      | 102    | 118   |
|                  | high      | 14    | 30   | 46    | 62   | 78  | 94      | 110    | 126   |
| White            | low       | 7     | 23   | 39    | 55   | 71  | 87      | 103    | 119   |
|                  | high      | 15    | 31   | 47    | 63   | 79  | 95      | 111    | 127   |

## Appendix G: Sample BAT/CMD Files

This section illustrates how batch files can be used to integrate Maximus with a third-party front end mailer.

### *Sample Waiting for Caller Batch File*

```

echo Off
rem * Insert your time zone here
set TZ=EST05

rem * Load FOSSIL driver
bnu

rem * OS/2 users only:
rem *
rem * Comment out the above call to BNU and uncomment
rem * the following MODE command. (This command should
rem * all be on one line.)
rem *
rem * mode com1:38400,n,8,1,,TO=OFF,XON=ON,IDSR=OFF,ODSR=OFF,
rem * OCTS=ON,DTR=ON,RTS=HS

rem * This is where you call Maximus itself. Change
rem * the '%1' and '%2' as necessary, to make Maximus
rem * work with your mailer. (OS/2 users should replace
rem * "max" with "maxp".)

:Loop
cd\Max
max -w
if errorlevel 50 goto event
if errorlevel 12 goto scan
if errorlevel 11 goto pack
if errorlevel 5 goto after
if errorlevel 4 goto error
if errorlevel 3 goto error
if errorlevel 2 goto after
if errorlevel 1 goto done
goto after

:event
rem * Run external maintenance program here.
goto Loop

:scan
rem * This command should invoke your scanner. For example:

squish out squash -fEchoToss.Log
scanbld user.bbs area.dat local matrix @echotoss.log
goto loop

:pack
rem * This should invoke your mail packer. For example:

squish squash
scanbld user.bbs area.dat local matrix

```

```

rem * (OS/2 users should replace "squish" with "squishp"
rem * and "scanbld" with "scanbldp".)
goto Loop

:after
rem * Insert after-caller utilities here.
goto Loop

:error
ECHO A fatal error occurred!

:done
ECHO Maximus down
exit

```

### ***Sample FrontDoor Batch File***

```

echo off
REM * Insert your time zone here
SET TZ=EST5

rem * Load FOSSIL driver
bnu

:loop
cd\FD
FD
if errorlevel 100 goto Local
if errorlevel 40 goto Maint
if errorlevel 34 goto UnpackMail
if errorlevel 33 goto B2400
if errorlevel 32 goto B1200
if errorlevel 31 goto B300
if errorlevel 10 goto Done
goto loop

:Local
rem * A local log-on to Maximus
cd \Max
Max -k
goto after_Max

:B2400
cd \Max
Max -b2400 -p1
goto After_Max

:B1200
cd \Max
Max -b1200 -p1
goto After_Max

:B300
cd \Max
Max -b300 -p1
goto After_Max

:After_Max
if errorlevel 12 goto scan
if errorlevel 11 goto pack
scanbld user.bbs area.dat local
goto loop

:unpackmail

```

```

rem * This should invoke your mail unpacker.

squish in out squash link -fEchoToss.Log
scanbld user.bbs area.dat @echotoss.log matrix
goto Loop

:scan
rem * This should invoke your mail scanner.

squish out squash -fechotoss.log
scanbld user.bbs area.dat local matrix @echotoss.log
goto loop

:pack
rem * This should invoke your mail packer.

squish squash
scanbld user.bbs area.dat local matrix

goto Loop

:maint
rem * Daily maintenance routine goes here
goto Loop
:done
ECHO FrontDoor ... down
exit

```

### Sample BinkleyTerm Batch File

```

echo off
rem * Insert your time zone here!
Set TZ=EDT5

:Top

rem * Unload and reload the FOSSIL driver and video FOSSIL.
rem * The following four lines can be omitted under OS/2.

VFOS_DEL
BNU -U
BNU
VFOS_BIO

rem * Start BinkleyTerm. Under OS/2, use "BTP unattended share"
rem * to ensure that the com port handle is properly passed to
rem * Maximus.

BT unattended
If ErrorLevel 255 goto Top
If ErrorLevel 96 goto BBS ; 9600 bps
If ErrorLevel 54 goto BBS ; 19200 bps
If ErrorLevel 30 goto Mail ; Incoming ARCmail/pkt/file
If ErrorLevel 24 goto BBS ; 2400 bps
If ErrorLevel 14 goto Maint ; Daily maintenance routine
If ErrorLevel 12 goto BBS ; 1200 bps
If ErrorLevel 3 goto BBS ; 300 bps
If ErrorLevel 2 goto Top
If ErrorLevel 1 goto End

:Mail
rem * Execute TOSS or IMPORT function here

```

```

squish in out squash link -fechotoss.log
scanbld user.bbs area.dat matrix @echotoss.log
goto Top

:Scan
rem * Execute SCAN and PACK functions here

squish out squash -fechotoss.log
scanbld user.bbs area.dat local matrix @echotoss.log
goto Top

:Pack
rem * Execute PACK functions here

squish squash
scanbld user.bbs area.dat local matrix
goto Top

:Maint
rem * Insert daily maintenance routine here
Goto Top

:BBS
rem * A human caller is here and wants into the BBS. Bink
rem * will create BBSBATCH.BAT/CMD which calls SPAWNBBS.BAT/CMD
rem * with the proper parameters, such as speed, time until
rem * next event, and port number). Maximus is invoked from
rem * SPAWNBBS.

c:
cd \binkley
bbsbatch
goto top

:End
rem * I exited Bink and back to DOS.
c:
cd \binkley
echo Binkley ... Down

```

### ***Sample BinkleyTerm SPAWNBBS.BAT***

```

echo OFF
cd \Max

rem * If running Max at a locked port rate, add a
rem * -s<speed> to the following command. For example,
rem * to lock the port at 38.4kbps, the following statement
rem * could be used: "max -b%2 -p%3 -t%4 -s38400". OS/2 users
rem * should do the same, but use "maxp" instead of "max".

max -b%2 -p%3 -t%4

:ELoop
If ErrorLevel 255 goto End
If ErrorLevel 65 goto Outside
If ErrorLevel 12 goto Export
If ErrorLevel 11 goto Mash
If ErrorLevel 10 goto End
If ErrorLevel 5 goto Acall
goto End

:Outside
call ERRORLVL.BAT

```



```
Max -r
goto ELoop

:Export
squish out squash -fechotoss.log
scanbld user.bbs area.dat local matrix @echotoss.log
goto end

:Mash
squish squash -fechotoss.log
scanbld user.bbs area.dat local matrix
goto end

:Acall
scanbld user.bbs area.dat local
goto end

:end
```

## Appendix H: Support Files

Table H.2 lists some of the hardcoded filenames that are used by Maximus:

**Table H.2 Hardcoded Filenames**

| Filename             | Description   |
|----------------------|---|
| <area>.dsc           | This file will be displayed to a user when entering an area, but only if the user's lastread pointer is set to 0. (For Squish areas only.)  |
| <area>.sqr           | This file is displayed to users every time they enter a message area. (For Squish areas only.)  |
| <areaname>.sqx       | This file is displayed to a user who attempts to enter a message in a read-only area. (Squish areas only.)  |
| active##.bbs         | This file is created by Maximus whenever a user logs onto node "##". The file is deleted when the user logs off.  |
| attrib.bbs           | This file is displayed to users who press a "?" at the attribute prompt in the full-screen message entry header.  |
| baduser.bbs          | Maximus will use this file (in the main system directory) to screen out unwanted names for new user logons. Maximus will display the <b>\max\misc\bad_user.bbs</b> file if a user attempts to use a name defined in this file. This file is a straight ASCII list of names not to be allowed on the BBS, one name to a line. Each name in the file is matched to either the first, last, or the entire name of the user. (If the first character of the string in baduser.bbs is a "~", Maximus will perform a substring search.) |
| blt-1.1,<br>blt-1.99 | Bulletins to be placed in QWK mail packets. These files should be placed in the <b>\max\olr</b> directory.  |
| browse.bbs           | The help file for the <b>Msg_Browse</b> command.  |
| chathelp.bbs         | The help file for the multinode chat.   |
| chatpage.bbs         | The file displayed to the user when a chat request is received from another node.   |
| chg_sent.bbs         | The file displayed when a user tries to edit a message which has already been sent, packed, or scanned out.   |
| chg_no.bbs           | The file displayed when a user tries to change a message which was written by someone else.   |
| descript.bbs         | This file is displayed to users who have a lastread pointer of 0 when they enter the message area containing this file. (For *.MSG areas only.)   |
| exbytes.bbs          | This file is displayed to users who attempt to download too many kilobytes in one session.  |
| excratio.bbs         | This file is displayed to users who attempted to download a file that would exceed the file download ratio.   |
| exctime.bbs          | This file is displayed to users who attempted to download a   |

|              |   |
|--------------|---|
|              | file that would exceed the time limit.  |
| file_bad.bbs | This file is displayed when an uploaded file fails the upload virus check. For more information, see <b>Upload Check Virus</b> in the system control file.  |
| file_ok.bbs  | This file is displayed when an uploaded file passes the upload virus check. For more information, see <b>Upload Check Virus</b> in the system control file.   |
| goodbye      | The file displayed to QWK users when they close the mail packet from your system. This file should be in the \max\olr directory.  |
| hello        | The file displayed to QWK users as they open the mail packet from your system. This file should be in the \max\olr directory.   |
| maxfiles.idx | This is the system-wide file index (as created by FB).  |
| mtag.dat     | This is a binary file used by Maximus to store message area tagging information   |
| names.max    | This file contains a list of aliases to be used when entering NetMail messages. This file must be in the \max directory. Each line has the following format:<br><br><alias>,<name>,<addr> [,<subj>]   |
|              | When Maximus spots a message addressed to <alias>, the message will be automatically readdressed to <name> with a destination address of <addr>. The optional <subject> can be used to enter the default for the message subject field. If a "*" is placed in front of <alias>, the alias definition can only be accessed by users in a class with the <b>MailFlags MsgAttrAny</b> attribute. |
| newfiles.dat | This file is displayed to QWK users when the user requests a listing of new files. This file should be placed in the \max\olr directory.  |
| news         | This file is displayed to a QWK user when the user requests the news file. This file should be placed in the \max\olr directory.  |
| notin.bbs    | Maximus displays this file when a user yells for the SysOp and the SysOp does not respond. This file should be in the \max\misc directory.  |
| rawdir.bbs   | If this file is placed in the Download path for a file area, it is automatically displayed to the user before the output of the <b>File_Raw</b> command.  |
| readonly.bbs | If this file exists in the directory for a read-only *.MSG message area, Maximus will display this file when the user attempts to enter a message.  |
| rules.bbs    | If this file exists in the directory for a *.MSG area, Maximus will display it file to all callers who enter the area.  |
| tag_file.bbs | This is the help file for the <b>File_Tag</b> command.  |

|              |   |
|--------------|---|
| tag_msg.bbs  | This is the help file for the <b>Msg_Tag</b> command.   |
| timeup.bbs   | This file is displayed to a user whose time limit has expired.  |
| why_ansi.bbs | This is the help file for the “ANSI graphics [Y,n,?]” question.   |
| why_fb.bbs   | This is the help file for the “Leave a message to the SysOp [Y,n,?]” question.  |
| why_fsed.bbs | This is the help file for the “Use MaxEd [Y,n,?]” question.   |
| why_hot.bbs  | This is the help file for the “Use Hotkeys [Y,n,?]” question.   |
| why_hu.bbs   | This is the help file for the “Goodbye [Y,n,?]” question.   |
| why_pc.bbs   | This is the help file for the “Use IBM Chars [Y,n,?]” question.   |
| why_pvt.bbs  | This is the help file for the “Private [Y,n]?” question when entering a message in TTY mode.                          |
| xpdate.bbs   | This file is displayed when a user's subscription expires by date. This file is in the <b>\max\misc</b> directory.    |
| xptime.bbs   | This file is displayed when a user's subscription expires by time. This file is in the <b>\max\misc</b> directory.    |
| yell.bbs     | Maximus will display this file (from the <b>\max\misc</b> directory) if a user tries to yell when yell is turned off. |

---

---

# Index

\*  
\*.MSG, 45, 46, 60, 393

9  
9600 bps, 31, 34, 337

A  
ACCEM, 115, 116  
access control file. *See* access.ctl  
access control string. *See* ACS  
access.ctl  
    Access, 415  
    Calls, 415  
    Cume, 415, 434  
    Desc, 415  
    End Access, 415  
    FileLimit, 415  
    FileRatio, 415, 418  
    Flags, 416  
    Flags DloadHidden, 416  
    Flags Hangup, 416  
    Flags Hide, 405, 414, 416  
    Flags NoFileLimit, 416  
    Flags NoLimits, 416  
    Flags NoTimeLimit, 416  
    Flags ShowAllFiles, 416  
    Flags ShowHidden, 416  
    Flags UploadAny, 416, 431  
    Key, 416  
    Level, 415, 416  
    LoginFile, 417  
    LogonBaud, 376, 384, 417  
    MailFlags, 417  
    MailFlags Editor, 374, 417  
    MailFlags LocalEditor, 374, 417  
    MailFlags MsgAttrAny, 376, 411, 417, 443  
    MailFlags NetFree, 417  
    MailFlags NoRealName, 417  
    MailFlags ShowPvt, 10, 412, 417  
    MailFlags WriteRdOnly, 10, 394, 417  
    OldPriv, 417  
    RatioFree, 415, 418  
    Time, 415, 418  
    UploadReward, 418  
    UserFlags, 418  
    XferBaud, 385, 418  
ACS, 55, 56, 64, 73, 83, 91, 93, 290, 335, 389, 390, 391, 395, 396, 402, 429  
Address, 361  
alias, 24  
ANS2BBS, 116, 117

ANS2MEC, 116, 117  
ANSI, 6, 7, 11, 17, 18, 25, 39, 59, 82, 101, 103, 105, 116, 117, 118, 127, 140, 143, 220, 237, 249, 250, 324, 337, 342, 367, 383, 404, 405, 427, 428, 429, 444  
ARC, 408  
ARCTIC, 33  
arrow keys  
    using menu options with, 402  
AUTOEXEC.BAT, 33, 34, 110, 128  
AVATAR, 7, 11, 17, 18, 25, 59, 82, 101, 105, 127, 140, 219, 220, 237, 249, 324, 327, 337, 342, 383, 399, 404, 405, 428, 436

B  
barricade, 71, 72, 235, 236, 390, 395, 429  
    extended, 72  
batch files  
    errorlevel, 98  
BinkleyTerm, 41, 439  
BNU, 2, 33, 34, 46, 437, 439  
BORED, 16, 18, 25, 143, 284, 381  
BUFFERS=, 36

C  
cable, 32  
callinfo.bbs, 103, 104  
CB chat, 112  
CD-ROM, 67, 68, 69, 70, 121, 377, 396, 397, 427  
chain.txt, 103, 104  
characters  
    high-bit, 25  
chat, 2, 9, 27, 28, 50, 107, 110, 111, 112, 132, 147, 241, 248, 253, 311, 345, 356, 365, 367, 382, 388, 403, 410, 434, 442  
colors control file. *See* colors.ctl  
colors.ctl  
    Popup Border, 387  
    Popup highlight, 387  
    Popup List, 387  
    Popup LSelect, 387  
    Popup Text, 388  
    Status Bar, 388  
    Status Chat, 388  
    Status Key, 388  
    WFC Activity, 388  
    WFC ActivityBor, 388  
    WFC Keys, 388  
    WFC KeysBor, 388  
    WFC Line, 389  
    WFC Modem, 388  
    WFC ModemBor, 388

- WFC Name, 389
- WFC Status, 389
- WFC StatusBor, 389
- colors.lh, 387
- COM.SYS, 32
- COM16550.SYS, 33
- compress.cfg, 366, 386
- conference areas, 9
- CONFIG.SYS, 34, 36, 113, 128
- control.dat, 387
- credit. *See* NetMail, credit
- custom menu, 73, 74, 340, 400
- CVTUSR, 38, 117, 118

## D

- data carrier detect. *See* DCD
- DCD, 31, 32, 245, 252, 260, 261, 433
- DESQview, 107, 355, 432
- DigiBoard, 33
- DIP switches, 31
- display files. *See* MECCA, files
- Display\_Menu, 406
- door. *See* external program
- door interface, 103, 104, 105
- door.sys, 103, 104
- doors
  - OS/2 and, 27
- dorinfo1.def, 103, 104
- DoubleDOS, 355, 432
- download, 21, 26, 28, 395, 408
- downloads
  - free, 66
  - staged, 68
- DTR, 359, 433
- duplicate file checking, 379

## E

- EchoMail areas, 9
- EDITCAL, 119
- editor
  - ASCII, 35, 37, 58, 66, 74
- EMS
  - swapping to, 357
- english.mad, 122, 123, 424
- errorlevel, 42, 43, 44, 45, 46, 50, 80, 98, 99, 100, 101, 129, 130, 361, 362, 365, 414, 423, 424, 430, 437, 438
- events
  - external, 49
  - yell, 49, 50
- exebbs.bat, 41
- expiration. *See* subscription
- expiration date, 87, 328
- extended ASCII, 25, 143, 393, 404
- extended barricades, 72
- external program, 8, 14, 16, 21, 27, 39, 42, 45, 46, 49, 76, 97, 98, 99, 101, 102, 103, 104, 105, 106, 124, 221, 243, 255, 256, 298, 299, 321, 332, 334, 348, 351, 354, 357, 367, 369,

- 377, 382, 384, 401, 406, 414, 421, 423, 427, 430

## F

- farea.dat, 38
- FB, 22, 69, 70, 119, 121, 379, 396, 397, 443
- FidoNet, 9, 39, 41, 110, 361, 362, 364
- file area control file. *See* filearea.ctl
- file areas, 37, 59, 61
  - hierarchical, 62
  - overriding, 23
  - searching, 20
- file attach, 11, 80, 81, 82, 235, 341, 356, 366, 373, 389, 393, 411
- filearea.ctl, 38
  - ACS, 395
  - Barricade, 71, 395, 429
  - Desc, 372, 395
  - Description, 395
  - Download, 61, 65, 68, 395, 397, 443
  - End FileArea, 395
  - FileArea, 61, 63, 67, 395
  - FileDivisionBegin, 63, 395
  - FileDivisionEnd, 63, 396
  - FileList, 68, 102, 121, 396
  - MenuName, 396
  - Override, 396
  - Type, 397
  - Type CD, 67, 397
  - Type DateAuto, 69, 369, 397
  - Type DateList, 69, 70, 119, 369, 397
  - Type DateManual, 369, 397
  - Type FileList, 120
  - Type Free, 397
  - Type FreeBytes, 397
  - Type FreeSize, 397
  - Type Hidden, 266, 373, 377, 397
  - Type NoNew, 397
  - Type Slow, 397
  - Type Staged, 377, 397
  - Upload, 62, 398
- files
  - deleting, 23
  - downloading, 21
  - moving, 23
  - tagging, 21, 22
- files.bbs, 68, 236, 396, 408
- FILES=, 36
- flow control
  - CTS, 360
  - DSR, 360
  - XON, 360
- FOSSIL, 2, 32, 33, 46, 104, 357, 360, 437, 438, 439
- FrontDoor, 41, 364, 439
- FSR, 26
- function keys
  - using menu options with, 402

## G

- GAP, 103

goto, 44, 45, 46, 47, 438, 439, 440, 441  
 guest account, 7

## H

Hayes, 3, 31, 40, 79, 358  
 HeaderFile, 398, 399  
 help levels, 24

## I

install.exe, 30  
 installation, 2, 29, 30, 33, 34, 37, 38, 45, 48,  
 108, 110, 111, 138, 353

## K

Kermit, 418, 419, 422

## L

label, 43, 44, 45, 47, 100, 115, 177, 228, 229,  
 322, 323, 338, 340  
 language, 26  
 language control file. *See* language.ctl  
 language.ctl, 76, 385  
     Language, 385, 424  
 lastread, 13, 102, 118, 126, 133, 135, 138, 141,  
 237, 314, 430, 442  
 lastread pointer  
     crosslinked, 118, 429  
 LIBPATH, 133  
 locked port, 35  
 logo.bbs, 432  
 LZH, 408

## M

MAID, 122  
 mailchecker, 128, 375, 411  
 mailer, 41, 45, 46, 48, 100, 106, 110, 356, 433,  
 437  
 marea.dat, 38  
 Master Control Program. *See* MCP  
 matrix  
     *See* Net, NetMail, 401  
 max.ctl, 37, 153  
     Address, 361  
     After Call, 365  
     After EchoMail, 361  
     After Edit, 362  
     After Local, 362  
     Alias System, 235, 365, 366  
     Answer, 79, 358, 359, 361  
     Area Change Keys, 365  
     Ask Alias, 365, 366  
     Ask Phone, 366  
     Attach Archiver, 81, 366  
     Attach Base, 366  
     Attach Path, 80, 366  
     Baud Maximum, 359  
     Busy, 358, 359, 433  
     Charset Chinese, 367  
     Charset Swedish, 367

Chat Capture, 367  
 Comment Area, 8, 14, 21, 60, 368, 410  
 Connect, 359  
 ContentsHelp, 382  
 Dos Close Standard Files, 354  
 Edit Disable, 368  
 FidoUser, 362  
 File Access, 354  
 File Callers, 251, 354  
 File Date, 368  
 File Date Automatic, 66  
 File Password, 354  
 FileData, 368  
 FileList Margin, 369  
 First File Area, 369  
 First Menu, 76, 370  
 First Message Area, 370  
 Format Date, 370, 373  
 Format FileFooter, 371  
 Format FileFormat, 371  
 Format FileHeader, 371  
 Format MsgFooter, 371  
 Format MsgFormat, 371  
 Format MsgHeader, 371  
 Format Time, 373  
 Gate NetMail, 362  
 Highest FileArea, 373  
 Highest MsgArea, 373  
 Init, 79, 359  
 Input Timeout, 373  
 Kill Attach, 81, 373  
 Kill Private, 374  
 Local Input Timeout, 375  
 Log EchoMail, 9, 61, 362, 394  
 Log File, 354  
 Log Mode, 354  
 Logon Level, 7, 52, 375  
 Logon Preregistered, 375  
 Logon TimeLimit, 375  
 Mailchecker Kill, 375  
 Mailchecker Reply, 375  
 Mask Carrier, 360  
 Mask Handshaking, 360  
 MaxMsgSize, 376  
 MCP Pipe, 113, 132, 355  
 MCP Sessions, 113, 355  
 Menu Path, 376  
 Message Edit, 15, 81, 362  
 Message Send Unlisted, 363, 364  
 Message Show, 95, 363  
 MessageData, 376  
 Min Logon Baud, 376, 384  
 Min NonTTY Baud, 376  
 Min RIP Baud, 376  
 Multitasker, 355, 432  
 Name, 355  
 No Critical Handler, 360  
 No Password Encryption, 38, 355  
 No RealName Kludge, 376, 394  
 No SHARE.EXE, 355  
 Nodelist Version, 364  
 Output, 360

- Path Inbound, 81, 356
- Path IPC, 111, 356
- Path Language, 356, 385
- Path Misc, 356
- Path NetInfo, 364
- Path Output, 356
- Path System, 357
- Path Temp, 357
- Reboot, 357
- Ring, 358, 361
- RIP Path, 345, 377
- Save Directories, 68, 377, 427
- Send Break to Clear Buffer, 361, 428
- Single Word Names, 5, 377
- Snoop, 357
- Stage Path, 68, 377, 397
- StatusLine, 377
- Strict Time Limit, 378
- Swap, 98, 357
- SysOp, 357
- Task, 358, 423, 433
- Track Base, 378
- Track Exclude, 378
- Track Modify, 92, 378
- Track View, 89, 379
- Upload Check Dupe, 67, 379
- Upload Check Dupe Extension, 379
- Upload Check Virus, 379, 443
- Upload Log, 380
- Upload Space Free, 380, 383
- Use UMSGIDs, 385
- Uses Application, 375, 381
- Uses BadLogon, 381
- Uses Barricade, 71, 72, 381
- Uses BeginChat, 381
- Uses BOREDHelp, 381
- Uses ByeBye, 381
- Uses Cant\_Enter\_Area, 381
- Uses Configure, 381
- Uses DayLimit, 382
- Uses EndChat, 382
- Uses EntryHelp, 382
- Uses FileAreas, 62, 328, 371, 382, 396
- Uses Filename\_Format, 382
- Uses HeaderHelp, 382
- Uses Leaving, 105, 382
- Uses LocateHelp, 382
- Uses Logo, 383
- Uses MaxEdHelp, 383
- Uses MsgAreas, 62, 328, 371, 383, 390
- Uses NewUser1, 383
- Uses NewUser2, 383
- Uses NoMail, 383
- Uses NoSpace, 380, 383
- Uses NotFound, 383
- Uses ProtocolDump, 383
- Uses Quote, 350, 384
- Uses ReplaceHelp, 384
- Uses Returning, 384
- Uses Rookie, 384
- Uses Shell\_Leaving, 384
- Uses Shell\_Returning, 384
- Uses TimeWarn, 384
- Uses TooSlow, 384
- Uses Tunes, 384
- Uses Welcome, 383, 384, 385
- Uses XferBaud, 385
- Video, 358
- Yell, 9
- Yell Off, 385
- max.prm, 35, 36, 38, 247, 432
- MaxEd, 16, 17, 18, 25, 39, 143, 237, 284, 340, 368, 374, 383, 404, 406, 444
- MaxPipe, 27, 124
- maxuapi.dll, 133
- MCP, 111, 112, 131, 355
- MECCA, 58, 59, 77, 82, 86, 87, 97, 98, 100, 101, 103, 104, 105, 115, 116, 117, 124, 125, 147, 148, 152, 153, 321, 322, 323, 324, 325, 326, 334, 335, 337, 344, 347, 348, 349, 357, 368, 383, 408, 411, 416, 428, 430
  - compiler, 58
  - files, 58
- MECCA token, 58, 87, 98, 101, 104, 105, 116, 152, 321, 322, 323, 344, 347, 416
  - [?below], 335
  - [?equal], 335
  - [?file], 335
  - [?line], 335
  - [access], 335
  - [accessfile], 335
  - [acs], 335
  - [acsfile], 335
  - [alist\_file], 328
  - [alist\_msg], 328
  - [ansopt], 332, 333
  - [ansreq], 333
  - [apb], 344
  - [b1200], 336
  - [b2400], 337
  - [b9600], 337
  - [bell], 326
  - [bg], 324
  - [black], 324
  - [blink], 324, 326
  - [blue], 324
  - [bright], 325
  - [brown], 324
  - [bs], 326
  - [chat\_avail], 345
  - [chat\_notavail], 345
  - [choice], 333
  - [city], 328
  - [ckoff], 347
  - [ckon], 347
  - [clear\_stacked], 334, 347
  - [cleol], 326
  - [cleos], 327
  - [cls], 74, 327
  - [col80], 337
  - [color], 337
  - [colour], 337
  - [comment], 347
  - [copy], 347



- [cr], 327
- [cyan], 324
- [darkgray], 324
- [date], 328
- [decimal], 347
- [delete], 347
- [dim], 325
- [dl], 328
- [dos], 348
- [down], 327
- [endcolor], 337, 342
- [endcolour], 337
- [endrip], 86, 337, 342, 343
- [enter], 333, 348
- [exit], 337, 350
- [expert], 337
- [expiry\_date], 328
- [expiry\_time], 328
- [fg], 325
- [file\_carea], 329
- [file\_cname], 329
- [file\_darea], 329
- [file\_sarea], 329
- [filenew], 337
- [first], 329
- [fname], 329, 333
- [got], 338
- [goto], 322, 323, 333, 340
- [gray], 324
- [green], 324
- [hangup], 348
- [hex], 348
- [hotkeys], 338
- [ibmchars], 348
- [ifentered], 331, 338
- [ifexist], 338
- [iffse], 338
- [iffsr], 338
- [ifkey], 336
- [iflang], 76, 77, 338
- [iftask], 339
- [iftime], 339
- [incity], 339
- [include], 348
- [islocal], 340
- [isremote], 340
- [jump], 340
- [key?], 348
- [key\_poke], 348, 349
- [keyoff], 336
- [keyon], 336
- [label], 340
- [language], 349
- [lastcall], 329
- [lastuser], 329
- [leave\_comment], 333, 368
- [left], 327
- [length], 329
- [lf], 327
- [lightblue], 324
- [lightcyan], 324
- [lightgreen], 324
- [lightmagenta], 324
- [lightred], 324
- [link], 337, 348, 349, 350, 430
- [load], 325
- [locate], 327
- [log], 349
- [magenta], 324
- [maxed], 340
- [menu], 101, 332, 333, 334
- [menu\_cmd], 77, 349
- [menupath], 349
- [mex], 152, 349
- [minutes], 329
- [more], 349
- [moreoff], 350
- [moreon], 350
- [msg\_attr], 340
- [msg\_carea], 329
- [msg\_checkmail], 350, 383, 411
- [msg\_cmsg], 329
- [msg\_cname], 330
- [msg\_conf], 341
- [msg\_darea], 330
- [msg\_echo], 341
- [msg\_fileattach], 81, 341
- [msg\_hmsg], 330
- [msg\_local], 341
- [msg\_matrix], 341
- [msg\_next], 342
- [msg\_nomsgs], 342
- [msg\_nonew], 342
- [msg\_noread], 342
- [msg\_nummsg], 330
- [msg\_prior], 342
- [msg\_sarea], 330
- [netbalance], 330
- [netcredit], 330
- [netdebit], 330
- [netdl], 330
- [newfiles], 349, 350, 408
- [no\_keypress], 342, 343
- [nocolor], 342
- [nocolour], 342
- [node\_num], 330
- [norip], 342
- [nostacked], 342, 343
- [notkey], 336
- [notontoday], 343
- [novice], 343
- [ofs], 343
- [on], 326
- [onexit], 350
- [open], 104, 332, 334
- [pause], 350, 428
- [permanent], 343
- [phone], 330
- [post], 104, 332, 334
- [priv\_abbrev], 335
- [priv\_desc], 335
- [priv\_down], 335
- [priv\_level], 335
- [priv\_up], 335

- [quit], 74, 337, 350
- [quote], 323, 350, 384
- [ratio], 331
- [readln], 331, 332, 334, 338, 345
- [realname], 331
- [red], 324
- [regular], 343
- [remain], 331
- [repeat], 350
- [repeatseq], 351
- [response], 331
- [right], 327
- [rip], 86, 343
- [ripdisplay], 87, 345, 377
- [riphasfile], 87, 343
- [rippath], 87, 345, 377
- [ripsend], 87, 295, 344, 345, 377
- [save], 325, 326
- [setpriv], 335
- [sopen], 334
- [steady], 326
- [store], 332, 333, 334
- [string], 351
- [subdir], 351
- [sys\_name], 331
- [syscall], 331
- [sysop\_name], 331, 357
- [sysopbell], 327
- [tab], 327
- [tag\_read], 351
- [tag\_write], 351
- [tagged], 344
- [textsize], 346
- [time], 331
- [timeoff], 331
- [top], 344
- [tune], 351
- [ul], 331
- [unsigned], 351
- [up], 328
- [user], 332
- [usercall], 321, 332
- [white], 324
- [write], 104, 332, 334, 430
- [xclude], 335
- [xtern\_dos], 97, 104, 351
- [xtern\_ervl], 98, 351
- [xtern\_run], 98, 104, 351
- [yellow], 323, 324
- menu options
  - linked, 74, 401, 406, 413
- MenuFile, 399
- menus
  - dynamic, 62
- menus control file. *See* menus.ctl
- menus.ctl, 38
  - Chat\_Answer, 27
  - Chat\_CB, 27, 403
  - Chat\_Page, 27, 403
  - Chat\_Pvt, 403
  - Chat\_Toggle, 28, 403
  - Chg\_Alias, 24, 403
  - Chg\_Archiver, 26, 28, 403
  - Chg\_City, 24, 403
  - Chg\_Clear, 404
  - Chg\_Editor, 404
  - Chg\_FSR, 26, 404
  - Chg\_Help, 24, 404
  - Chg\_Hotkeys, 25, 404
  - Chg\_IBM, 25, 404
  - Chg\_Language, 26, 76, 123, 349, 404
  - Chg\_Length, 25, 404
  - Chg\_More, 25, 404
  - Chg\_Nulls, 24, 405
  - Chg\_Password, 24, 405
  - Chg\_Phone, 24, 405
  - Chg\_Protocol, 26, 28, 405
  - Chg\_Realname, 405
  - Chg\_RIP, 86, 405
  - Chg\_Tabs, 25, 405
  - Chg\_Userlist, 8, 26, 405
  - Chg\_Video, 25, 405
  - Chg\_Width, 24, 406
  - Clear\_Stacked, 406
  - Conf, 400
  - Ctl, 400
  - Display\_File, 76, 102, 153, 284, 406
  - Display\_Menu, 75, 76, 111, 401, 406
  - Echo, 400
  - Edit\_Abort, 18, 406
  - Edit\_Continue, 17, 19, 406
  - Edit\_Delete, 19, 406
  - Edit\_Edit, 18, 384, 407
  - Edit\_From, 18, 19, 407
  - Edit\_Handling, 18, 19, 407
  - Edit\_Insert, 19, 407
  - Edit\_List, 18, 407
  - Edit\_Quote, 19, 407
  - Edit\_Save, 18, 407
  - Edit\_Subj, 17, 19, 407
  - Edit\_To, 17, 19, 407
  - End Menu, 398
  - File\_Area, 20, 373, 397, 408
  - File\_Contents, 23, 382, 408
  - File\_Download, 21, 383, 408
  - File\_Hurl, 23, 62, 408
  - File\_Kill, 23, 408
  - File\_Locate, 20, 67, 373, 382, 408
  - File\_NewFiles, 69, 408
  - File\_Override, 23, 408
  - File\_Raw, 23, 408, 443
  - File\_Tag, 22, 408, 443
  - File\_Titles, 20, 409
  - File\_Type, 409
  - File\_Upload, 383, 409
  - File\_View, 21
  - Goodbye, 8, 14, 381, 409
  - HeaderFile, 398, 399
  - Key\_Poke, 403, 409
  - Leave\_Comment, 357, 410
  - Link\_Menu, 284, 406, 410, 414
  - Local, 401
  - Matrix, 401
  - Menu, 398, 399

- MenuColor, 399
- MenuFile, 73, 398, 399, 400
- MenuLength, 399, 400
- MEX, 153, 284
- Msg\_Area, 11, 373, 394, 410
- Msg\_Browse, 13, 14, 373, 410, 412, 442
- Msg\_Change, 12, 411
- Msg\_Checkmail, 411
- Msg\_Current, 411
- Msg\_Download\_Attach, 81, 411
- Msg\_Edit\_User, 411
- Msg\_Enter, 10, 11, 411
- Msg\_Forward, 15, 411
- Msg\_Hurl, 16, 411
- Msg\_Kill, 15, 375, 412
- Msg\_Kludges, 90, 95, 363, 412
- Msg\_Reply, 12, 375, 392, 412
- Msg\_Reply\_Area, 412
- Msg\_Restrict, 412
- Msg\_Tag, 14, 28, 412, 443
- Msg\_Track, 89, 91
- Msg\_Unreceive, 412
- Msg\_Upload, 15, 83, 84, 375, 376, 412, 427
- Msg\_Upload\_QWK, 28, 413
- Msg\_Xport, 16, 413
- NoCLS, 401
- NoDsp, 75, 401
- NoRIP, 87, 401
- OptionWidth, 400
- Press\_Enter, 74, 413
- Read\_Current, 13
- Read\_DiskFile, 413
- Read\_Individual, 413
- Read\_Next, 11, 413, 414
- Read\_Nonstop, 12, 413
- Read\_Original, 13, 413, 414
- Read\_Previous, 11, 413, 414
- Read\_Reply, 13, 413, 414
- ReRead, 105, 401
- Return, 410, 414
- RIP, 87, 401
- Same\_Direction, 414
- Title, 400
- Track Base, 91
- Track Exclude, 91
- Track Modify, 91
- Track View, 91
- User\_Editor, 27, 414
- Userlist, 8, 414
- UsrLocal, 401
- UsrRemote, 402
- Version, 74, 414
- Who\_Is\_On, 9, 345, 365, 414
- Xterm\_DOS, 97, 401, 414
- Xterm\_Erlvl, 98, 99, 106, 284, 414, 433
- Xterm\_Run, 98, 401, 414
- Yell, 8, 414, 423
- message area control file. *See* msgareactl
- message areas, 37, 59
- EchoMail, 9, 42, 45, 46, 47, 48, 61, 89, 94, 122, 128, 129, 130, 235, 341, 355, 361, 362, 363, 391, 393, 411
- hierarchical, 62
- Local, 9, 12, 129, 362, 393, 394, 401
- NetMail, 9, 12, 15, 16, 18, 19, 39, 42, 45, 46, 47, 81, 82, 85, 122, 129, 238, 341, 356, 362, 363, 364, 394, 411, 417, 443
- packing, 48
- read-only, 10
- renumbering, 48
- message tracking system. *See* MTS
- messages
  - anonymous, 10
  - audit trail, 90, 95
  - downloading, 83
  - editing, 18
  - forwarding, 15
  - printing, 16, 413
  - priority, 90
  - private, 10
  - public, 10
  - quoting, 17, 19
  - renumbering, 125
  - tracking, 88
  - uploading, 83
- MEX, 1, 103, 147, 155, 398, 399, 425
  - arguments
    - pass-by-reference, 186, 215
  - arrays, 190, 215
    - accessing, 191
    - as function parameters, 193
    - declaring, 190
  - blocks, 211
  - casts, 206
    - printing using, 207
  - comments, 149, 211
  - compiler, 151
  - directive
    - #include, 164, 165, 184, 211, 224
  - directives, 164
  - escape sequences, 150, 161
  - function, 149
  - functions, 212
    - arguments, 184
    - arguments for, 149
    - calling, 149, 179
    - definitions, 183
    - prototypes, 183, 213
    - return values, 187, 213
  - language files and, 425
  - statement
    - compound, 214
    - do while, 175
    - for, 176, 214
    - goto, 177
    - if, 172
    - return, 151, 180
    - while, 174
  - strings, 150, 162, 213
    - assigning, 162
    - indexing, 163

- structures, 198, 216
  - as arguments, 204
  - declaring variables as, 200, 216
  - defining types, 198
  - types, 212
  - variables, 159
    - declaring, 212
- MEX directives
  - #include, 148
- MEX functions
  - ansi\_detect, 249, 250
  - call\_close, 247, 250
  - call\_numrecs, 247, 251, 252
  - call\_open, 247, 250, 251, 252, 354
  - call\_read, 247, 251
  - carrier, 245, 252, 261
  - chat\_querystatus, 240, 248, 253
  - chatstart, 248, 253
  - class\_abbrev, 248, 254
  - class\_info, 248, 254, 255, 256
  - class\_loginfile, 248, 256
  - class\_name, 248, 257
  - class\_to\_priv, 248, 257
  - close, 243, 258
  - compressor\_num\_to\_name, 246, 258
  - create\_static\_data, 248, 258, 259, 260, 261, 271, 297
  - create\_static\_string, 248, 259, 260, 261, 272, 297
  - dcd\_check, 245, 253, 260
  - destroy\_static\_data, 248, 261
  - destroy\_static\_string, 248, 261
  - display\_file, 243, 263
  - Display\_Menu, 284
  - do\_more, 242, 263, 264
  - file\_area, 246, 265
  - fileareafindclose, 246, 265, 266
  - fileareafindfirst, 246, 265, 266, 267
  - fileareafindnext, 246, 266, 267
  - fileareafindprev, 246, 266, 267
  - fileareaselect, 219, 246, 267
  - filecopy, 243, 267
  - filedate, 241, 243, 268
  - fileexists, 243, 268, 269
  - filefindclose, 244, 269, 270
  - filefindfirst, 244, 269, 270
  - filefindnext, 244, 269, 270
  - filesize, 241, 243, 269, 270, 271, 294
  - get\_static\_data, 248, 271, 297
  - get\_static\_string, 248, 272, 297
  - getch, 220, 243, 272, 280, 281
  - hstr, 248, 273, 282
  - input\_ch, 220, 233, 234, 243, 273, 274, 276, 278, 280
  - input\_list, 220, 233, 234, 243, 274, 275, 276, 280
  - input\_str, 220, 233, 234, 243, 277, 280, 317
  - iskeyboard, 243, 279
  - issnoop, 242, 279
  - itostr, 245, 279, 280, 283, 313
  - kbhit, 243, 280, 281
  - keyboard, 243, 280
  - language\_num\_to\_name, 246, 281
  - language\_num\_to\_string, 281
  - localkey, 243, 281
  - log, 247, 281
  - long\_to\_stamp, 244, 282
  - lstr, 248, 282
  - ltostr, 245, 283, 313
  - mdm\_command, 245, 283
  - mdm\_flow, 245, 283
  - menu\_cmd, 243, 284
  - msg\_area, 246, 284
  - msgareafindclose, 246, 285
  - msgareafindfirst, 246, 285, 286, 287
  - msgareafindnext, 246, 286, 287
  - msgareafindprev, 246, 286, 287
  - msgareaselect, 219, 246, 287
  - open, 243, 258, 287, 291, 292, 295, 310, 317
  - print, 149, 170, 242, 288
  - privok, 248, 290
  - prm\_string, 247, 291
  - protocol\_num\_to\_name, 246, 291
  - read, 243, 291, 292, 310
  - readln, 243, 292, 310, 332, 333, 338
  - remove, 243, 292
  - rename, 243, 293
  - reset\_more, 242, 263, 264, 293
  - rip\_detect, 249, 293, 294
  - rip\_hasfile, 87, 249, 294
  - rip\_send, 87, 249, 294
  - screen\_length, 238, 242, 295
  - screen\_width, 238, 242, 295
  - seek, 243, 295
  - set\_output, 242, 296
  - set\_static\_data, 248, 271, 297
  - set\_static\_string, 248, 272, 297
  - set\_textsize, 242, 298, 310
  - shell, 97, 98, 243, 298
  - sleep, 245, 299
  - snoop, 242, 299
  - stamp\_string, 244, 300
  - stamp\_to\_long, 218, 244, 300
  - strfind, 244, 300, 301
  - stridx, 244, 301, 304
  - strlen, 214, 244, 302, 318
  - strlower, 244, 302
  - strpad, 244, 302, 303
  - strpadleft, 244, 303
  - strridx, 244, 301, 303, 304
  - strtoi, 245, 304
  - strtok, 244, 305
  - strtol, 245, 305, 306
  - strtrim, 244, 306, 307
  - strupper, 244, 307
  - substr, 244, 307, 308
  - tag\_dequeue\_file, 246, 308
  - tag\_get\_name, 246, 308
  - tag\_queue\_file, 221, 246, 308, 309
  - tag\_queue\_size, 246, 308, 309
  - tell, 243
  - term\_length, 238, 242, 310

- term\_width, 238, 242, 310
  - time, 244, 282, 310
  - time\_check, 245, 311
  - timeadjust, 245, 311
  - timeadjustsoft, 245, 311, 312
  - timeleft, 245
  - timeon, 245
  - timestamp, 245
  - uitostr, 245, 279, 313
  - ultostr, 245, 283, 313
  - usercreate, 247
  - userfilesize, 247, 314
  - userfindclose, 247, 314
  - userfindnext, 247, 315
  - userfindopen, 247, 314, 315, 316
  - userfindprev, 247, 314, 315
  - userfindseek, 247, 316
  - userremove, 247, 316
  - userupdate, 247, 316
  - vidsync, 234, 242, 317
  - write, 243, 310, 317
  - writeln, 243, 310, 318
  - xfertime, 245, 318
  - MEX globals
    - farea, 235
    - id, 234
    - input, 233
    - mare, 235
    - msg, 236
    - sys, 239
    - usr, 236
  - MEX structs
    - \_callinfo, 241
    - \_cstat, 240
    - \_date, 240
    - \_farea, 235
    - \_ffind, 241
    - \_instancedata, 234
    - \_mare, 235
    - \_msg, 236
    - \_stamp, 240
    - \_sys, 239
    - \_time, 240
    - \_usr, 236
  - modem, 1, 3, 5, 30, 31, 32, 33, 34, 35, 39, 40, 41, 79, 80, 103, 245, 252, 273, 283, 284, 358, 359, 360, 361, 388, 428
    - V.32, 34
    - V.34, 33, 34
    - V.FC, 33, 34
  - modem cable. *See* cable
  - modem status register. *See* MSR
  - more prompt, 25
  - MR, 48, 125, 126, 392, 393
  - msgareactl, 38
    - ACS, 60, 61, 389
    - AttachPath, 81, 366, 389
    - Barricade, 71, 390, 429
    - Desc, 60, 61, 372, 390
    - Description, 390
    - End MsgArea, 59, 389, 390
    - FileDivisionEnd, 63
    - MenuName, 390, 427
    - MsgArea, 59, 63, 389, 390
    - MsgDivisionBegin, 63
    - MsgDivisionEnd, 63, 391
    - Origin, 391
    - Override, 391
    - Owner, 91, 93, 94, 392, 393
    - Path, 60, 392
    - Renum Days, 126, 392
    - Renum Max, 126, 393
    - Style, 60, 393
    - Style \*.MSG, 60, 393
    - Style Alias, 365, 393
    - Style Anon, 393
    - Style Attach, 80, 81, 82, 356, 393
    - Style Audit, 91, 94, 392, 393
    - Style Conf, 393, 400
    - Style Echo, 393, 400, 427
    - Style HiBit, 393
    - Style Hidden, 373, 394
    - Style Loc, 394
    - Style Local, 394, 401
    - Style Net, 81, 394, 401
    - Style NoMailCheck, 394
    - Style NoNameKludge, 10, 393, 394
    - Style Pub, 60, 394
    - Style Pvt, 60, 394
    - Style ReadOnly, 394, 417
    - Style RealName, 365, 394
    - Style Squish, 60, 393, 394
    - Style Tag, 394
    - Tag, 61, 129, 362, 372
  - MSR, 360
  - MTS, 88, 89, 90, 91, 92, 93, 96, 378, 392, 393
    - QWK packets and, 93
    - reports, 90, 92
  - multinode, 107
- ## N
- NetMail
    - credit, 9, 39, 417
    - QWK packets and, 85
  - NoAccess, 55
  - odelist, 363, 364
  - Novell, 36
- ## O
- off-line reader, 28
  - off-line reader control file. *See* readerctl
  - Opus, 33, 117, 118, 418, 419, 420, 421, 422
  - OpusComm, 2, 33, 34
  - ORACLE, 59, 117, 126, 127
  - origin line, 391, 427
  - OS/2, 1, 3, 25, 27, 29, 30, 32, 33, 35, 37, 39, 40, 41, 42, 46, 50, 65, 80, 97, 106, 107, 108, 109, 110, 111, 112, 124, 128, 131, 133, 148, 224, 235, 299, 356, 358, 359, 360, 379, 433, 437, 439

## P

packer, 28, 46, 47, 75, 82, 93, 410. *See also* offline reader  
 PAK, 408  
 Path NetInfo, 364  
 phone number, 6, 39, 101, 105, 141, 145, 203, 237, 330, 366, 381, 387, 435  
 PID, 393  
 printing  
   messages, 413  
 printing messages, 16  
 privilege level, 52  
 protocol, 26  
   Opus-compatible, 419, 421  
 protocol control file. *See* protocol.ctl  
 protocol.ctl, 422  
   ControlFile, 419  
   DescriptWord, 419  
   DownloadCmd, 419  
   DownloadKeyword, 420  
   DownloadString, 419, 420, 421  
   End Protocol, 420  
   FilenameWord, 420  
   LogFile, 421  
   Protocol, 421  
   Type Batch, 421  
   Type Bi, 421  
   Type Opus, 419, 421  
   UploadCmd, 421  
   UploadKeyword, 419, 420, 421, 422  
   UploadString, 419, 421, 422

## Q

questionnaire, 321, 332, 334  
 QuickBBS, 103, 104, 117, 118  
 quote, 12, 17, 19, 85, 89, 94, 161, 162, 226, 350, 374  
 QWK, 14, 26, 75, 82, 83, 84, 85, 89, 93, 386, 387, 403, 410, 412, 413, 427, 442, 443

## R

ratio  
   download, 51  
 RBBS, 103, 104  
 reader control file. *See* reader.ctl  
 reader.ctl  
   Archivers, 386  
   Max Messages, 75, 387  
   Packet Name, 75, 386  
   Phone Number, 75, 387  
   Work Directory, 386  
 real name, 10, 102, 331, 365, 393, 394  
 result codes, 31, 80  
 reward, 418  
 REXX, 133, 134  
 REXX API  
   MaxLoadFuncs, 134  
   MaxUnloadFuncs, 134  
   UserFileClose, 134  
   UserFileCreateRecord, 135

UserFileFind, 135

UserFileFindClose, 136

UserFileFindNext, 136, 137, 138

UserFileFindOpen, 136, 137

UserFileFindPrior, 137

UserFileGetNewLastread, 135, 138, 141

UserFileInitRecord, 135, 138

UserFileOpen, 134, 138, 139

UserFileSize, 139

UserFileUpdate, 139

RIPscrip, 6, 11, 17, 25, 26, 39, 85, 86, 87, 125, 143, 237, 242, 249, 294, 295, 298, 337, 342, 343, 344, 345, 346, 347, 367, 368, 376, 377, 382, 399, 400, 401, 405

runbbs.bat, 46, 48, 50, 99, 109, 122

runbbs.cmd. *See* runbbs.bat

runfb.bat, 121

## S

Save Directories, 68, 377

SCANBLD, 45, 46, 128, 129, 130

scanner, 47, 437, 439

screen writes, 234

SEALink, 21, 144

security, 38, 51, 82, 356

SEEN-BY, 391

serial cards

  intelligent, 33, 107

Session Monitor. *See* SM

SHARE.EXE, 355

shell, 27

SILT, 37, 38, 68, 93, 130, 131, 152, 353, 354, 390, 395, 396, 397, 417

SILT directive

  Include, 353

  Version14, 353

  Version17, 353

SIO, 33, 106

SM, 131

source file

  MEX, 148

spawnbbs.bat, 41

SQPACK, 48

Squish, 42, 45, 46, 60, 362, 385, 386, 392, 393

squish.cfg, 392, 393

startup.cmd, 110

subscription, 87, 328

swapping, 98

system control file. *See* max.ctl

## T

tab characters, 25

tear line, 393

TheDraw, 117

TRACKEXP, 95, 96

TRACKIMP, 96

translation characters, 8, 14, 21, 101, 102, 104, 283, 332, 334, 347, 348, 349, 351, 370, 373, 406, 410

TSR, 33

TTY, 25, 101, 127, 140, 237, 324, 376, 405, 444

## U

Upload, 15, 22, 28, 62, 84, 375, 379, 380, 398, 409, 412, 413  
    reward, 37  
user editor, 14, 27, 39, 54, 88, 411, 414, 433, 435  
user file  
    creating, 38  
    encryption, 38  
Uses Leaving, 382

## V

Version, 8, 364

## W

WFC, 45, 46, 49, 50, 79  
WildCat!, 103  
Windows, 355  
    Windows 95, 36, 110  
    Windows for Workgroups, 36  
WWIV, 103, 104

## X

X00, 2, 33, 34  
Xmodem, 21, 22, 144  
XMS  
    swapping to, 357

## Z

ZIP, 66, 408  
Zmodem, 2, 21, 144, 145, 422, 433





# Maximus System Operations Manual

— |

| —

— |

| —

# Maximus System Operations Manual

Version 3.0

Title: LOGONAME.EPS from  
Creator:  
CreationDate: Sat Jul 08 13:34:15 1995

Lanius Corporation, Kingston, Ontario, Canada  
Copyright © 1989, 1995 by Lanius Corporation. All rights reserved.

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system without the prior written permission of Lanius Corporation.

Maximus and Squish are trademarks of Lanius Corporation.

MS-DOS, Windows, Windows 95 and Windows NT are trademarks of Microsoft Corporation.

IBM, PC-DOS and OS/2 are trademarks of International Business Machines Corporation.

Hayes is a trademark of Hayes Microcomputer Products, Inc.

V.FC is a trademark of Rockwell International.

MD5 is a trademark of RSA Data Security, Inc.

BNU is a trademark of Unique Computing Pty Ltd.

X00 and SIO are trademarks of Raymond L. Gwinn.

All other trade names are trademarks of their respective holders.

Printed in Canada.

10 9 8 7 6 5 4 3 2 1

---

# Contents

|  |           |
|--|-----------|
| <b>1. Introduction.....</b>                                    | <b>1</b>  |
| 1.1. About Maximus.....  | 1         |
| 1.2. System Requirements .....                                 | 2         |
| 1.3. Typographical Conventions.....                            | 3         |
| <b>2. Maximus Overview .....</b>                               | <b>5</b>  |
| 2.1. Waiting for Callers.....                                  | 5         |
| 2.2. Logging On.....   | 5         |
| 2.3. Command Stacking.....                                     | 6         |
| 2.4. Guest Accounts .....                                      | 7         |
| 2.5. The Main Menu .....                                       | 7         |
| 2.6. The Message Section.....                                  | 9         |
| 2.7. The File Menu.....  | 20        |
| 2.8. The Change Setup Menu .....                               | 24        |
| 2.9. The SysOp Menu.....                                       | 27        |
| 2.10. The Chat Menu .....                                      | 27        |
| 2.11. The Off-Line Reader Menu .....                           | 28        |
| <b>3. Installation.....</b>                                    | <b>31</b> |
| 3.1. Step 1: Unpacking the Distribution Files .....            | 31        |
| 3.2. Step 2: Running the Installation Program.....             | 32        |
| 3.3. Step 3: Configuring your Modem .....                      | 33        |
| 3.4. Step 4: Installing Communications Drivers .....           | 34        |
| 3.5. Step 5: Editing Configuration Files .....                 | 37        |
| 3.6. Step 6: About Control Files .....                         | 39        |
| 3.7. Step 7: Compiling the Control Files .....                 | 39        |
| 3.8. Step 8: Starting Maximus.....                             | 40        |
| 3.9. Step 9: Support for Remote Callers.....                   | 41        |
| 3.10. Step 10: Miscellaneous Information.....                  | 50        |
| <b>4. Customization.....</b>                                   | <b>53</b> |
| 4.1. Events and Yelling.....                                   | 53        |
| 4.2. Access Control: Classes, Privilege Levels and Locks ..... | 55        |
| 4.3. Display Files: *.mec and *.bbs .....                      | 62        |
| 4.4. Message Areas and File Areas.....                         | 63        |
| 4.5. Maintaining File Areas .....                              | 69        |
| 4.6. Barricades and Extended Barricade Files.....              | 75        |
| 4.7. Menus .....   | 78        |
| 4.8. QWK Mail Packing.....                                     | 80        |
| 4.9. Multilingual Support.....                                 | 81        |

|   |            |
|---|------------|
| <b>5. Maximus Subsystems.....</b>                 | <b>83</b>  |
| 5.1. Waiting for Callers Subsystem .....          | 83         |
| 5.2. Local File Attaches.....                     | 84         |
| 5.3. QWK Mail Packer .....                        | 86         |
| 5.4. RIPscrip Support.....                        | 89         |
| 5.5. User Expiration and Subscriptions.....       | 92         |
| 5.6. Message Tracking System .....                | 92         |
| <b>6. External Programs .....</b>                 | <b>103</b> |
| 6.1. Execution Methods.....                       | 103        |
| 6.2. Swapping .....                               | 104        |
| 6.3. Errorlevel Batch Files.....                  | 104        |
| 6.4. Restarting After Errorlevel Exit.....        | 105        |
| 6.5. External Program Translation Characters..... | 107        |
| 6.6. Running Doors.....                           | 109        |
| 6.7. On-Line User Record Modification.....        | 111        |
| 6.8. Doors and OS/2.....                          | 112        |
| <b>7. Multinode Operations .....</b>              | <b>113</b> |
| 7.1. Installation .....                           | 114        |
| 7.2. Multinode Chat Operation.....                | 117        |
| 7.3. Master Control Program.....                  | 119        |
| <b>8. Utility Documentation .....</b>             | <b>121</b> |
| 8.1. ACCEM: MECCA Decompiler .....                | 121        |
| 8.2. ANS2BBS/MEC: ANSI to MEC conversion.....     | 122        |
| 8.3. CVTUSR: User File Conversions .....          | 123        |
| 8.4. EDITCAL: Call Modification Utility.....      | 124        |
| 8.5. FB: File Database Compiler .....             | 125        |
| 8.6. MAID: Language File Compiler.....            | 128        |
| 8.7. MAXPIPE: OS/2 Redirection Utility .....      | 130        |
| 8.8. MECCA: Display File Compiler.....            | 131        |
| 8.9. MR: Maximus Renumbering Program.....         | 131        |
| 8.10. ORACLE: Display File Viewer.....            | 133        |
| 8.11. SCANBLD: *.MSG Database Builder .....       | 134        |
| 8.12. SILT: Control File Compiler .....           | 137        |
| 8.13. SM: Session Monitor .....                   | 138        |
| <b>9. REXX User File Interface.....</b>           | <b>141</b> |
| 9.1. Introduction .....                           | 141        |
| 9.2. Function Descriptions .....                  | 142        |
| 9.3. Accessing "usr." Variables .....             | 148        |
| <b>10. Introduction to MEX Programming.....</b>   | <b>155</b> |
| 10.1. About MEX.....                              | 155        |
| 10.2. MEX Road Map .....                          | 156        |
| 10.3. Creating a Sample MEX Program .....         | 156        |

|  |            |
|--|------------|
| 10.4. Compiling MEX Programs.....                                | 159        |
| 10.5. Running MEX Programs.....                                  | 160        |
| <b>11. MEX Language Tutorial.....</b>                            | <b>163</b> |
| 11.1. Program Development Cycle.....                             | 163        |
| 11.2. Elements of a MEX Program.....                             | 165        |
| 11.3. Variable Declarations.....                                 | 167        |
| 11.4. Variable Scope.....  | 172        |
| 11.5. Preprocessor.....  | 172        |
| 11.6. Calculations and Arithmetic.....                           | 173        |
| 11.7. Displaying Output.....                                     | 179        |
| 11.8. Flow Control.....  | 181        |
| 11.9. Function Calls.....  | 187        |
| 11.10. Arrays.....   | 199        |
| 11.11. Structures.....   | 207        |
| 11.12. Casts.....  | 216        |
| 11.13. Further Explorations in MEX.....                          | 218        |
| <b>12. MEX for C and Pascal Programmers.....</b>                 | <b>221</b> |
| 12.1. Comments.....  | 221        |
| 12.2. Include Files.....   | 221        |
| 12.3. Blocks.....  | 221        |
| 12.4. Function Definitions.....                                  | 222        |
| 12.5. Types.....   | 222        |
| 12.6. Variable Declarations.....                                 | 222        |
| 12.7. Function Prototypes.....                                   | 223        |
| 12.8. Function Return Values.....                                | 223        |
| 12.9. Strings.....   | 224        |
| 12.10. Compound Statements.....                                  | 224        |
| 12.11. Arithmetic, Relational and Logical Operators.....         | 224        |
| 12.12. The for Statement.....                                    | 225        |
| 12.13. Arrays.....   | 225        |
| 12.14. Pointers.....   | 225        |
| 12.15. Pass-By-Reference Arguments.....                          | 226        |
| 12.16. Variable-Length Arrays.....                               | 226        |
| 12.17. Structures.....   | 226        |
| 12.18. Run-Time Library Support.....                             | 227        |
| <b>13. Interfacing MEX with Maximus.....</b>                     | <b>229</b> |
| 13.1. User Information.....                                      | 229        |
| 13.2. Message Area Information.....                              | 230        |
| 13.3. File Area Information.....                                 | 230        |
| 13.4. Changing Message and File Areas.....                       | 231        |
| 13.5. Displaying Output.....                                     | 231        |
| 13.6. Retrieving Input.....                                      | 232        |
| 13.7. File I/O.....  | 233        |
| 13.8. Menu Commands, Displaying Files and External Programs..... | 233        |

|  |            |
|--|------------|
| 13.9. Download Queue .....                       | 233        |
| 13.10. Other Functions .....                     | 233        |
| <b>14. MEX Compiler Reference .....</b>          | <b>235</b> |
| 14.1. Command Line Format .....                  | 235        |
| 14.2. Environment Variables .....                | 236        |
| 14.3. Error Messages and Warnings .....          | 236        |
| <b>15. MEX Library Reference.....</b>            | <b>245</b> |
| 15.1. Global Variables and Data Structures ..... | 245        |
| 15.2. Functions by Category.....                 | 254        |
| 15.3. Function Descriptions.....                 | 262        |
| <b>16. MEX Language Reference.....</b>           | <b>333</b> |
| 16.1. Operator Precedence.....                   | 333        |
| 16.2. Language Grammar .....                     | 333        |
| <b>17. MECCA Language Reference .....</b>        | <b>335</b> |
| 17.1. Usage Guide.....                           | 335        |
| 17.2. Color Token Listing .....                  | 337        |
| 17.3. Cursor Control and Video Tokens .....      | 340        |
| 17.4. Informational Tokens .....                 | 342        |
| 17.5. Questionnaire Token Listing .....          | 346        |
| 17.6. Privilege Level Controls.....              | 349        |
| 17.7. Lock and Key Control .....                 | 350        |
| 17.8. Conditional and Flow Control Tokens.....   | 351        |
| 17.9. Multinode Tokens .....                     | 359        |
| 17.10. RIPscrip Graphics .....                   | 360        |
| 17.11. Miscellaneous Token Listing .....         | 361        |
| <b>18. Control File Reference.....</b>           | <b>367</b> |
| 18.1. SILT Directives.....                       | 367        |
| 18.2. System Control File.....                   | 367        |
| 18.3. Language Control File.....                 | 400        |
| 18.4. Off-Line Reader Control File.....          | 400        |
| 18.5. Colors Control File.....                   | 401        |
| 18.6. Message Area Control File .....            | 404        |
| 18.7. File Area Control File .....               | 409        |
| 18.8. Menu Control File.....                     | 413        |
| 18.9. Access Control File .....                  | 430        |
| 18.10. Protocol Control File .....               | 434        |
| 18.11. Event File.....                           | 438        |
| 18.12. Language Translation File Reference.....  | 439        |
| <b>Appendices .....</b>                          | <b>443</b> |
| Appendix A: Common Problems.....                 | 443        |
| Appendix B: Error Messages.....                  | 445        |



**Contents ix**

|  |            |
|--|------------|
| Appendix C : Command Line Switches.....  | 448        |
| Appendix D: Local Keystrokes.....        | 450        |
| Appendix E: User Editor Keystrokes ..... | 451        |
| Appendix F: AVATAR Colors .....          | 452        |
| Appendix G: Sample BAT/CMD Files .....   | 453        |
| Appendix H: Support Files .....          | 458        |
| <b>Index.....</b>                        | <b>461</b> |



Bookmark page (not for publication)

Version: 3.0 (300)

VersionBold **3.0 (300)**

VariableName:

- The name must be from 1 to 32 characters long.
- The name is case-sensitive. This means that “delta,” “Delta,” “DELTA,” and “DeLtA” refer to four distinct objects.
- Names can include letters and underscores. Names can also include digits, except in the first character of the name. (This means that “top10” is a valid name, whereas “7up” is not.)

ForDOS: DOS only!

ForOS2: OS/2 only!

Note: 

|            |
|------------|
| Title: Aff |
| Creator:   |

Warning: 

|      |
|------|
| Titl |
| Cre  |