MacIvory User's Guide

Preface

MacIvory provides the benefits of symbolic processing along with the features of a Macintosh personal computer. Before using your MacIvory, read "Introduction to MacIvory Machines" (this will familiarize you with the system). Then, read the user documentation supplied by Apple, and the disk drive and monitor manufacturers, so that you will feel confident about using the Macintosh computer, and the MacIvory system's components.

Once you are familiar with your MacIvory, its components, and the Macintosh user interface, return to this manual. Begin by reading and following the steps in "Getting Started with MacIvory" to position and power up your MacIvory System.

This book is divided into two main sections: a user's manual and a programmer's reference. The user's manual can help you to get started using MacIvory, and provides information on some common Macivory operations. The programmer's reference provides information on how to use features of the Macintosh operating system from Genera and other topics for programmers who are developing MacIvory applications.

You may also want to refer to this additional documentation:

- For information about adapting programs written on Symbolics 3600-series machines to MacIvory, see the section "Porting Genera Applications to Ivory Machines".

- For information about the Genera 8.0 software and how it differs from Genera 7.4 Ivory, see *Genera 8.0 Release Notes*. For information about Symbolics machines and the Genera software development environment, refer to the books contained within the Genera documentation set. New Symbolics users should refer to these titles first:

  - *Genera Workbook*

  - *Site Operations*

  - *Genera User's Guide*

Overview of MacIvory

The MacIvory system makes Symbolics' Genera software environment, which enables the comprehensive development, prototyping, and delivery of applications, available to Apple Macintosh II users.

Whenever possible, MacIvory maintains the Macintosh user interface paradigm for video, I/O processing and communications. The MacIvory board set plugs directly into the Macintosh II with no other system modifications required. Genera applications can be "opened" under MacOS just like any other Macintosh window.

MacIvory software basically consists of two parts: Code that runs on the Ivory processor and code that runs on the Macintosh. The two processors communicate with each other through a special region of memory that they both access. MacIvory has exclusive use of a memory region allocated to it by the Macintosh when the Macintosh is powered on or restarted, and the Macintosh uses the rest of memory. All input/output is handled through the Macintosh.

On the Ivory side, the code provided (written in Lisp) consists of:

- A Remote Procedure Call (RPC) interface.
- Genera modified as necessary to use the MacOS for all input/output, including keyboard, mouse, disk, Ethernet, and console screen accesses.
- IFEP code modified similarly. (The IFEP is the part of Genera software involved with booting and initializing a world. Historically, this code resided in read-only memory on the Front End Processor of a Symbolics machine; therefore the name IFEP, with I standing for Ivory.)

On the Macintosh side, the code provided (written in C) consists of:

- Code to initialize the MacIvory and the IFEP.
- Life-Support code required to support MacIvory.
- A Macintosh application that allows users to access applications running on Ivory.

Macintosh/Ivory Communication

Communication is based on a shared memory, readable and writable by both the guest (Ivory) and the host (Macintosh) processors. Outside of shared memory, each processor has its own memory space. This memory space is allocated to each processor when the Macintosh II is booted.

There is no shared disk space: The disk is partitioned into Ivory space and Macintosh space. (Ivory disk space is further partitioned into FEP file space and Lisp-Machine File Space just as it would be on a 3600 or XL400, but this is transparent to application users.)

There are two levels of communication between the Macintosh and Ivory:

- In shared memory, a buffered channel exists between drivers on the Macintosh and on Ivory for those operations requiring maximum performance, such as disk I/O and network communication.
- A remote procedure call (RPC) interface for general-purpose input/output.

The host (Macintosh running MacOS) is primary in the sense that it is booted

first and the guest (Ivory), looks like just one of the several application programs that the user can run. A Macintosh application program runs when the user wants to make use of Ivory. This program boots Ivory, and gives Ivory control over the keyboard, mouse and all or part of the screen. After that, the Ivory runs continuously, staying up between visits by the user, unless the user explicitly reboots the Ivory or reboots the Macintosh.

Communication between guest and host applications is managed by the Remote Procedure Call (RPC) protocol. The same protocol also allows Ivory to access the MacOS, and vice versa.

Software Components of MacIvory

In addition to standard Genera, software components for MacIvory include:

- RPC, the Remote Procedure Call facility
- The remote window facility built on RPC that allows the Genera window system to use the screen, keyboard, and mouse of the host Macintosh instead of directly manipulating a 3600-series machine's own local screen, keyboard, and mouse
- A user interface system that is adaptable to Genera style or Macintosh style
- Interface to the Macintosh toolbox
- Support for accessing Macintosh files
- Support for Macintosh peripherals, such as Ethernet, tape and serial lines
- Support for some Macintosh data and file formats

User's Manual for MacIvory

Introduction to MacIvory Machines

MacIvory machines integrate the Genera software environment with the Apple Macintosh II (including its operating system and the Toolbox) this way:

- The Genera software environment accelerates programmer productivity and provides the ability to do complex, real-world problem solving.
- The Macintosh offers a stylish and easy-to-learn user interface, along with access to a wide variety of third-party applications.

MacIvory provides full integration of these two systems; it is designed to combine the advantages of each, while delivering the benefits of both.

MacIvory relies on the Macintosh processor for I/O services such as keyboard, display, and disk access. All of Macintosh's system and toolbox entry points can be called from Lisp using Remote Procedure Calls (RPC). Additionally, Genera facilities (Dynamic Windows, for example) use Macintosh facilities, and MacIvory makes full use of the Macintosh for graphics. This kind of integration means that existing Genera applications — developed on the Symbolics 3600 series of machines — can readily take advantage of Macintosh features.

MacIvory enables you to design applications that employ either the Macintosh user interface (UI), the Genera user interface, or both. Furthermore, without changing your program, you can allow end users to make their own UI choices. MacIvory doesn't mandate a user interface style for your applications; you are free to create your own.

MacIvory Software Description

MacIvory software consists of:

Genera 8.1
: The Genera environment adapted to run on the Ivory processor; both a development and a delivery version are available. This software is contained within the Symbolics Distribution World and is distributed on disk and QIC-100 tapes.

MacIvory Support Software
: To provide MacIvory with the basic I/O services it needs to run Genera, and to facilitate high-level communication between the two processors by using a Remote Procedure Call (RPC) mechanism. This software is distributed on floppy diskettes.

Getting Started with MacIvory

Symbolics Service Personnel will configure and install your MacIvory System. After it is installed, perform these tasks:

1. Following each manufacturer's directions to ensure that you leave appropriate clearance for cooling, position the Macintosh computer and the external disk drive (if your system is equipped with one) conveniently in your workspace.

   Position the 19" monitor on either side of the Macintosh II or on a stand over the Macintosh II, as the manufacturer recommends.

   Warning: Be sure to position the 19" monitor so that there is at least an inch between the monitor and the Macintosh II to ensure adequate ventilation and avoid overheating. Do not place the 19" monitor near heat sources such as radiators or in an area subject to direct sunlight.

2. Make sure that the Symbolics keyboard template has been placed over the keys.

3. Power up (plug in and turn on) the external disk drive (if your system is equipped with one) and the Macintosh computer *according to each manufacturer's instructions*.

Note: At power up, you will momentarily see the Apple "Happy Mac" on your Macintosh screen, indicating that the Macintosh is starting up from the disk drive. You will also see a symbol in the lower left-hand corner of your screen that indicates the presence of pre-installed Ivory "life-support" software on the disk drive.

Controlling the Ivory Coprocessor on a MacIvory

When you first power on the Macintosh and start up an Ivory application (by double-clicking on the Genera icon, for example), the Ivory coprocessor is initialized, and Genera is cold booted. This may take a few minutes.

Once the Ivory coprocessor has been initialized and Genera has been cold booted (by the first Ivory application that needs it), you can open and quit that application — and other Ivory applications — as quickly and easily as you do ordinary (non-Ivory-dependent) Macintosh applications.

When you restart the Macintosh by choosing Restart from the Special menu (available from the Finder), this initializes the Macintosh processor, but preserves the state of the Ivory coprocessor's virtual memory.

Because the Ivory coprocessor's virtual memory is unaffected when you restart the Macintosh computer, the Ivory coprocessor does not need initializing — and Genera does not need cold booting — each time you restart the Macintosh computer.

When you open an Ivory application, it checks to see whether the Ivory coprocessor can be warm booted (that is, whether the Ivory coprocessor's virtual memory has some state), or whether it must be cold booted. If the Ivory coprocessor can be warm booted, you have two options:

1. Click on "Restart Lisp" when the alert box appears. This opens the application without initializing the Ivory coprocessor or cold booting Genera. Clicking on "Restart Lisp" preserves the contents of Ivory's virtual memory but warm boots Ivory, resetting its processes and closing its network connections.

2. Click on "Cold Boot Lisp" when the alert box appears. This opens the application, initializes the Ivory coprocessor, and cold boots Genera. Clicking on "Cold Boot Lisp" means the contents of Ivory's virtual memory will be lost.

If there is not enough state available to warm boot the Ivory coprocessor, then the application itself causes the Ivory coprocessor to initialize, and Genera to cold boot.

More information is available about cold booting and warm booting. See the section "Cold Booting". See the section "Warm Booting".

Using the Ivory Menu

Once you have opened an Ivory application and it is running, you can control it and the Ivory coprocessor by using the Ivory menu in the menu bar at the top of your Macintosh screen.

Tables 1 and 2 describe the Ivory menu items by showing you their Symbolics 3600-series machine equivalents.

Ivory Menu Item (invoked while running Lisp)	**3600-Series Machine Equivalent**
Restart Lisp	Halt Machine (Command Processor Command) Start (FEP Command)
Cold Boot Lisp	Halt Machine (Command Processor Command) Press RESET on the machine's front panel Hello (FEP Command) Boot (FEP Command)
Transfer to FEP	Halt Machine (Command Processor Command)
Restart FEP	Halt Machine (Command Processor Command) Reset FEP (FEP Command)
Cold Boot FEP	Halt Machine (Command Processor Command) Press RESET on the machine's front panel
Shut Down	Halt Machine (Command Processor Command) Shutdown (FEP Command)
Hide/Show Cold Load	No equivalent; causes the Cold Load Stream window to move back or forward.
Enable/Disable Network	No equivalent; selects whether Ivory (Enable) or the Macintosh (Disable) has access to the Ethernet cable.

Table 1. The Meaning of Ivory Menu Items (while Lisp is Running)

Using the Genera Application on a MacIvory

The Genera application is a program that provides MacIvory users with the traditional Symbolics user interface to the Genera software environment. Like other third-party supplied Macintosh applications, the Genera application can be invoked with the Macintosh user interface.

Ivory Menu Item (invoked from the FEP)	3600-Series Machine Equivalent
Transfer to Lisp	Continue (FEP Command)
Restart Lisp	Hello (FEP Command) if flod files need scanning Start (FEP Command)
Cold Boot Lisp	Press RESET on the machine's front panel Hello (FEP Command) Boot (FEP Command)
Restart FEP	Reset FEP (FEP Command)
Cold Boot FEP	Press RESET on the machine's front panel
Shut Down	Shutdown (FEP Command)
Hide/Show Cold Load	No equivalent; causes the Cold Load Stream window to move back or forward.
Enable/Disable Network	No equivalent; selects whether Ivory (Enable) or the Macintosh (Disable) has access to the Ethernet cable.

Table 2. The Meaning of Ivory Menu Items (while in the FEP)

Start up the Genera application by double-clicking on the Genera icon. The Genera application lets your MacIvory emulate a Symbolics 3600-series machine's console. On a MacIvory, however, the Genera window system can use all — or just part — of the available screen.

You can use the Macintosh's user interface software (for example, MultiFinder) to select between different Macintosh applications, including Genera. (For information about how to use the Macintosh system software, refer to the Apple user documentation supplied with your MacIvory system.)

Note: If you cannot find the mouse cursor while in Genera, press the Power On key (at the upper right-hand corner) on the Apple Extended Keyboard. This will cause the cursor to reappear at the Apple symbol on the Menu bar.

Caution: The Ivory Breath of Life Application (BOL FEP) which you see in your MacIvory Applications folder is used for initializing FEP file systems only. Experimenting with it while Lisp is running damages your Lisp system in such a way that you cannot warm boot. **Do not experiment with it.**

Accessing the Macintosh File System

Files on the MacIvory's Macintosh file system can be accessed from Genera using pathnames that follow Macintosh syntactic conventions.

To access the Macintosh file system from Genera on the local MacIvory, use this format:

HOST:*Volume*:*Folder*:*Folder*:*Filename*

In this format,

- The word HOST is literal; it refers to the local Macintosh file system, similar to the way the word FEP refers to the local FEP files system on 3600-series machines.
- *Volume* is the Macintosh volume name (the name that's displayed under the disk icon on the Macintosh desktop).
- *Folder* names the directory (or subdirectory) in which the file *Filename* resides.

To access the MacIvory's Macintosh file system from Genera on a remote MacIvory or Symbolics 3600-series machine, use this format (note the addition of a vertical bar in the pathname):

MacIvory-Host-Name|HOST:*Volume*:*Folder*:*Folder*:*Filename*

Here,

- *MacIvory-Host-Name* names the MacIvory host whose Macintosh file system you want to access.

The Macintosh syntax uses colons to separate pathname components. To show a directory on the local Macintosh file system:

```
Command: Show Directory (files [default Y:>Dodds>pending>*.*.newest])
HOST:dsk:zippy:pinhead:*


HOST:dsk:zippy:pinhead:*
    3292 free, 64733/68025 used (blocks of 4608 8-bit bytes)
    (773 files in 154 dirs)

        complex          5    640(8)         11/03/89 22:49:40
        simple           5    640(8)         11/03/89 23:39:58
        test-file.sab    5   1250(8)         11/03/89 22:54:59

  15 blocks in 3 files
```

Macintosh Pathname Completion

Genera's completion facilities can access the file systems of various hosts, including Macintosh computers. To perform Macintosh pathname completion, Genera looks in the Macintosh's file system and returns a new, possibly more specific string than the one that you have entered, expanding any unambiguous abbreviations in a Macintosh host-dependent fashion.

Insert an asterisk (*) to get wildcard possibilities (including devices) within Macintosh pathnames. Use the c-/ and c-? keystrokes to see multiple possibilities for what you have already typed. For more information about using these keystrokes, see the section "c-? and c-/". For information about quoting special characters (*, for example), see the section "Macintosh Pathname Quoting".

Macintosh Pathname Quoting

To the Macintosh, all characters are legal in pathnames and only colons delimit directories. This means, for example, that asterisks may appear as a component of a Macintosh pathname. Since Genera considers asterisk to be a wildcard, you cannot type a Macintosh pathname containing an asterisk without quoting it. Therefore, when typing a pathname such as

HOST:*volume*:*Graphics:picture

you need to quote the asterisk to indicate to Genera that it is not a wildcard. Circle-Plus (⊕, sy-sh-+) is the quote character. So you would actually type:

Host:*volume*:⊕*Graphics:picture

This also means that when Genera prints a Macintosh pathname that has an asterisk as part of its name, it quotes the asterisk using the Circle-Plus character.

The Symbolics Keyboard for MacIvory

The Symbolics keyboard is optimized for Lisp programmers, and for running Genera. For information about installing a Symbolics keyboard for a MacIvory, see the section "Installing a Symbolics Keyboard and Three-Button Mouse on a MacIvory". For information about using the Symbolics keyboard, see the section "Index of Special Function Keys".

For more information, see the section "Using the Symbolics Keyboard with Native Macintosh Applications".

Changing the Keyboard Mappings on a MacIvory

Changing the Shift Key Order (Control, Meta, Shift) on Apple Keyboards

You can change the shift key order on an Apple Keyboard following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Common Settings*.

3. Click on the appropriate shift key order.

Reconfiguring Keyboard Mappings

You can reconfigure the settings on either the Apple or Symbolics keyboard following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Keyboard Control*.

The keyboard control menu appears enabling you to reconfigure your keyboard mappings (see Figure !).

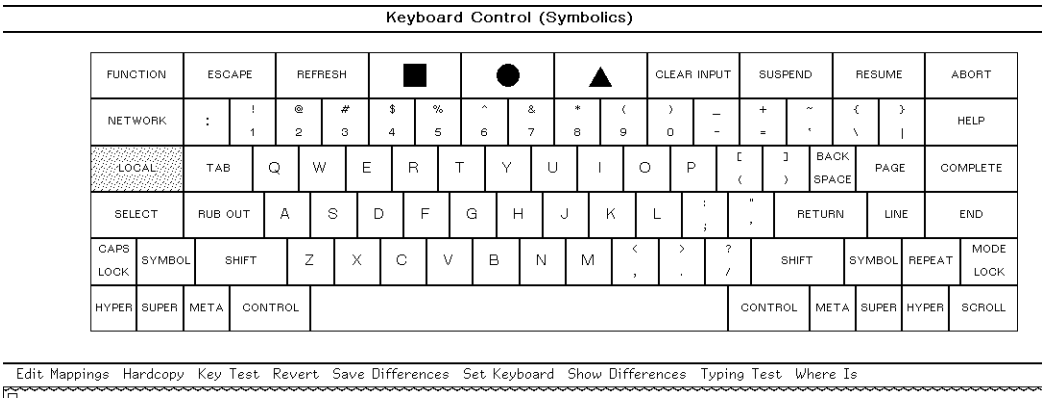

Figure 90. Keyboard Control Menu

You can reconfigure your keyboard mappings by clicking Left on *Edit Mappings* and specifying *All* at the prompt. The keyboard mappings for your keyboard appears as in Figure !.

You can change the keyboard mapping for each key by clicking Left on the current setting for the key and specifying a new setting. For example, you can change the current setting of the key sequence *Symbol A* from *Unassigned* to *S*:

1. Click Left on *Unassigned*.

2. Specify *S*.

3. Press RETURN

You can now obtain the letter *S* by pressing *Symbol A*. You can change this keyboard mapping again following the preceding steps. You can also use the Keyboard Control menu for:

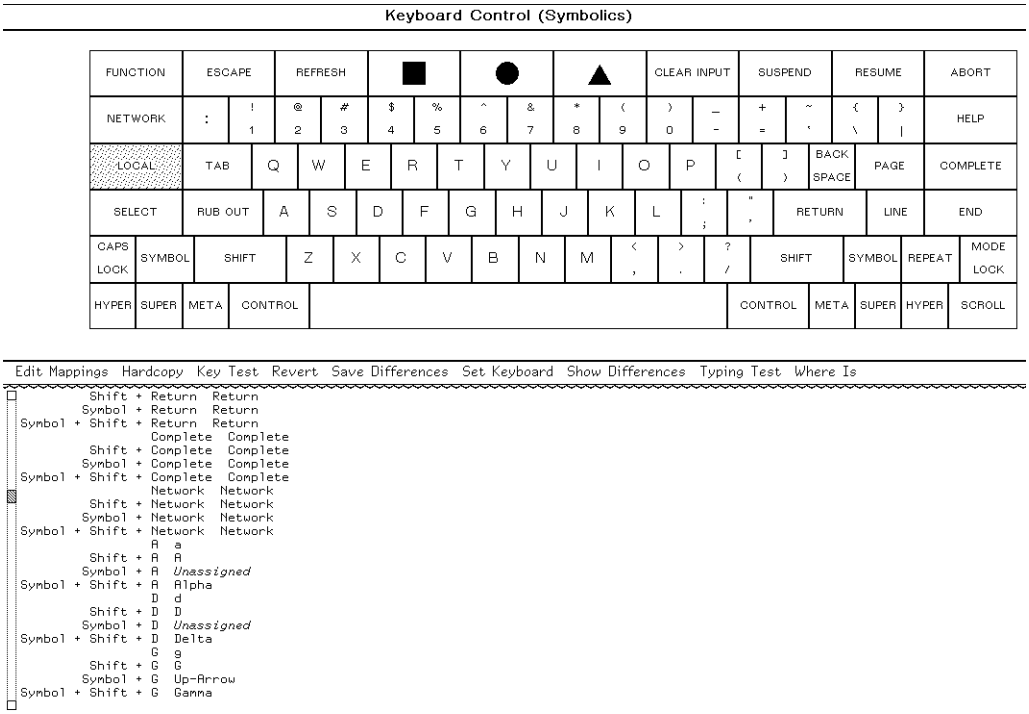

Figure 91. Reconfiguring Keyboard Mappings

- Hardcopying the Keyboard Layout
- Displaying Raw Keyboard and Mouse Transitions
- Resetting the Keyboard Mapping to a Standard Mapping
- Saving the Differences in Keyboard Mappings
- Setting the Keyboard to a Default or Predefined Keyboard
- Showing the Differences in Keyboard Mappings
- Displaying Key Mappings
- Determining the Key Mapping that Generates a Character

Hardcopying the Keyboard Layout

You can hardcopy your current keyboard layout and mappings following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Keyboard Control*.

3. click Left on *Hardcopy* for hardcopying the keyboard layout using default printer settings. You can customize the printer settings by clicking Right on *Hardcopy* and specifying:

    - The printer on which you are printing the keyboard mappings.
    - The print orientation (Portrait or Landscape).
    - Whether you include legends or mappings.
    - Whether you include codes (Octal, Decimal, or Hex).

Displaying Raw Keyboard and Mouse Transitions

You can determine whether your keyboard operates correctly by using the *Key Test* option. This option displays the key sequence generated when you press a key. You can start the *Key Test* option following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Keyboard Control*.

3. Click Left on *Key Test*.

Resetting the Keyboard Mapping to a Standard Mapping

You can set your customized keyboard mappings to a standard keyboard mapping following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Keyboard Control*.

3. Click Left on *Revert*.

Saving the Differences in Keyboard Mappings

You can place the differences between customized keyboard mappings and standard keyboard mappings on the kill ring following these steps:

1. Select *Keyboard* from the Options menu.

2. Select *Keyboard Control*.

3. Click left on *Save Differences*.

Setting the Keyboard to a Default or Predefined Keyboard

You can set your keyboard to either the default Symbolics keyboard or to another predefined keyboard following these steps:

1. Select *Keyboard* from the Options menu.
2. Select *Keyboard Control*.
3. You can set your keyboard to the default by clicking Left on *Set Keyboard*, or you can specify another predefined keyboard by clicking Right on *Set Keyboard*.

Showing the Differences in Keyboard Mappings

You can show the differences in mappings between your keyboard and the standard keyboard following these steps:

1. Select *Keyboard* from the Options menu.
2. Select *Keyboard Control*.
3. Click left on *Show Differences*.

Displaying Key Mappings

You can display the mappings for keys on your keyboard following these steps:

1. Select *Keyboard* from the Options menu.
2. Select *Keyboard Control*.
3. Click left on *Typing Test*.

You can now press any key and see the current keyboard mapping on your screen.

Determining the Key Mapping that Generates a Character

You can determine how a character is generated following these steps:

1. Select *Keyboard* from the Options menu.
2. Select *Keyboard Control*.
3. Click left on *Where Is*.
4. Specify a character.

The system displays the key sequence necessary for generating the character you specify.

Using the Print Spooler with Genera

To spool an LGP2/LGP3 printer from a MacIvory machine, you need one 8-pin-male-to-25-pin-male cable with a null modem in it. Alternatively, you can use any combination of cables that provides you with this configuration.

1. Plug the printer into one of the serial ports on the back of the Macintosh computer. You can run the printer from the modem port (`UNIT 0`) or the printer port (`Unit 1`).

2. Edit the printer's Namespace Object as follows:

```
Interface: SERIAL
Interface Options: UNIT n BAUD 9600 PARITY :none NUMBER-OF-DATA-BITS 8
        XON-XOFF-PROTOCOL yes
User Property: DTR-VALID nil
```

   More information is available about editing the printer's namespace object. See the section "Attributes for Objects of Class "Printer"". See the section "Using the Namespace Editor".

3. Set the baud rate to 9600 on the LGP2/LGP3 printer.

Note: If you need more information about Macintosh serial ports, check the Unit specification conventions in the MacIvory User's Guide.

For additional information about setting up the print spooler and the LGP2/LGP3 printer,see the section "Installing a Printer".

See the section "Uncrating the Printer".

See the section "Specifying the Switch Settings".

See the section "Loading the Hardcopy and Printer Support Tape".

See the section "Registering a Printer".

Note: If you boot the MacIvory while the print spooler is running, you must then restart the Macintosh computer. Otherwise, the serial stream is left open (and attempts to restart the print spooler will fail with an "unable to access the serial stream" error).

Using an Appletalk Printer From the Macintosh and Genera

- Printing via AppleTalk is supported only for Postscript printers.

- Printing via AppleTalk can not be used with our Print Spooler.

To use an AppleTalk printer do the following:

1. Select your target printer on the Macintosh side through the Chooser.

2. Create a printer object in the namespace that identifies your MacIvory as the host to which the printer is connected.

3. Set the default printer for your MacIvory to the printer object you just created. You can do this either in the namespace to make the effect permanent or with the Set Printer CP command.

You can now use the printer from Lisp using the normal hardcopy facilities of Genera.

The use of separate printer objects is required even if several MacIvories are using the same AppleTalk printer. Of course, each printer object in the namespace must have a unique name and pretty name. Therefore it is a good idea to include the name of the MacIvory in the printer's names.

For example,

Showing PRINTER SOUR-CREAM-LOS-ANGELES-TIMES in namespace SCRC:

```
Type: LGP2
Site: SCRC
Pretty Name: The Los Angeles Times for Sour Cream
Interface: AppleTalk
Host: SOUR-CREAM
Printer Location: SCRC 2 Outside KMP's office
```

The `Type` attribute should be LGP2, LGP3, or POSTSCRIPT depending on the type of printer. LGP2 corresponds to the old LaserWriter and LaserWriter Plus. LGP3 corresponds to the newer LaserWriter IINT and LaserWriter IINTX. LGP3 also works for Apple's latest printer, the Personal LaserWriter NT. Use POSTSCRIPT for non-Apple Postscript printers (for example, HP).

In the namespace object for the MacIvory, make sure that the `BITMAP-PRINTER` attribute is empty. Otherwise, Lisp tries to invoke the Print Spooler, which does not work.

If your Macintosh is running MultiFinder, it's a good idea to enable background printing in the Chooser when you select your AppleTalk printer. Otherwise, whenever you print something in Lisp, Genera stops while printing takes place. (You see the familiar Macintosh modal dialog boxes describing the print process appear in front of the Genera screen.)

Setting the Size of Application Icons That Use Ivory

If you use MultiFinder on the Macintosh, each application program icon has a memory size that you can control. Icons that give access to the Ivory coprocessor make extensive use of dynamic memory. Because their memory usage depends on your machine configuration, and because there is a tradeoff between memory consumption and performance, it can be to your advantage to adjust icon memory size to suit your particular needs. If you run Finder rather than MultiFinder on the Macintosh, application icons always use the full size of memory and you need not worry about this issue.

Icons that use the Macintosh window system, such as the Mac Dex example, can run in 256K and will run quite effectively in as little as 300K. Icons that use the Genera window system, such as the Genera icon, can run in 256K but will perform noticeably better with more memory. Genera uses large amounts of dynamic memory for fonts, window bit arrays, and auto-load RPC servers. The recommended memory allocation for good performance is 250K plus four times the size of your screen, plus one more screen size if backup screens are enabled. You can determine the size in K of your screen by evaluating this form:

```
(values tv:(ceiling (* (sheet-width main-screen) (sheet-height main-screen)
        (screen-bits-per-pixel main-screen))
        8192))
```

If you have plenty of memory on your Macintosh, you can set the size larger than this and get a small further improvement in performance, especially if you use a lot of large windows. If your Macintosh has a limited amount of memory or you want to save a lot of memory for running other applications at the same time as Genera, you can set the size of the Genera icon smaller, but switching windows will be slower.

You can set the size of an icon with the Configure MacIvory Application command or by using the procedure described in Apple documentation for setting the size of an application.

Configure MacIvory Application Command

Configure MacIvory Application

Use the Configure MacIvory Application command with no arguments. It presents a menu of fields which you must fill out and then press End.

The menu presents these fields:

From file — An existing Ivory-using Macintosh application; you can use Genera. For example, if Genera is on the desktop you can use the file name HOST:*disk*:Genera, where *disk* is the name of your Macintosh's hard disk.

To file — The name of a Macintosh file in which to place the output; it should have the same name as your **define-remote-program-framework**.

Application — The name of your **define-remote-program-framework** (use the pretty name).

Version — A Macintosh-style version number for your application, for instance 1.0d0.

Agent — Leave this set to Emb (which means that the application expects to find an embedded Ivory processor).

Initial application command — Set this to a command line that invokes a command defined by

your application that should be executed to start it up. Or you can leave this set to None, if the application should simply wait for a command (for example, from the menu bar) when started up.

If the host has more than one monitor, you can specify the number of the monitor (display screen) that will be used by Genera, using the **:screen-number** argument. These are the same numbers displayed in the Monitors section of the Macintosh Control Panel. Specifying "default" means the main monitor, the one with the menu bar on it. This is the default if you do not specify this argument.

If you leave the *Initial application command* field set to None, rather than supplying a Start Screen command, when Genera is started it will display a dialog box allowing interactive input of the arguments to the Start Screen command. The fields displayed in the dialog box depend on the hardware configuration, just like the command arguments noted above.

The arguments accepted by the Start Screen command, after the first few, vary depending on the available hardware. If you wish to specify these arguments you should run the Configure MacIvory Application command on the same machine where you will run the resulting icon, or on a machine with a similar hardware configuration.

If the host has color display hardware, you can specify whether the Genera window system should run in color mode or in monochrome mode, using the **:multi-bit-screen** argument; see the section "Basic Color Support in MacIvory".

Controlling the Genera Screen on a MacIvory

Controlling the Size and Placement of a Genera Screen on a MacIvory

You can use Backup screens by using the Enable/Disable Backup screen option from the Macintosh Options menu. This option enables you to place one screen on top of another and to use each screen individually.

You can use Movable screens by using the Enable/Disable Movable screen option from the Macintosh Options menu. This option enables you to adjust the size of the Genera screen in the same manner that you adjust the size of a Macintosh screen.

Using the MacIvory Control Panel

The MacIvory control panel lets you easily perform various maintenance functions for the MacIvory system. It is integrated as part of the Apple control panel. You can access the Macintosh control panel by clicking on Control Panel from the Apple pull-down menu and then scrolling through the left-hand column to the MacIvory icon.

The control panel's three window panes describe these areas:

- Hardware information
- Status and configuration options
- Disk maintenance and manipulation

Figure ! displays the initial MacIvory control panel.

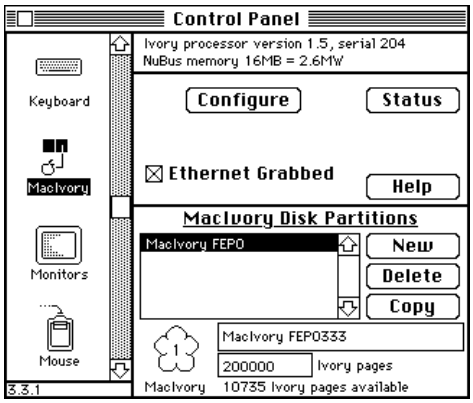

Figure 92. The MacIvory Control Panel

Hardware status

Describes the Ivory processor and the NuBus memory that it uses, if they are present. Memory size is specified both as raw size in megabytes (MB) and usable size in megawords (MW) of 40 bits each plus 8 error correction bits. Additional hardware configuration details are available from Genera commands such as Show Herald and Show Machine Configuration.

Status and configuration options

The middle pane displays these command buttons:

[Configure], which brings up a dialog that you can use to examine and change several software configuration parameters. Most users will never need to change these parameters. The [Help] button in the dialog explains how to use it. Figure ! displays this dialog.

[Ethernet Grabbed], which is checked if an Ethernet interface exists and the Ivory is using it. If this box is not checked, the Macintosh can use the Ethernet interface if one exists. Clicking on this box switches control of the Ethernet interface between the Ivory and the Macintosh. This change takes effect immediately.

[Status], which brings up a scrollable window containing the current status of communications between the Ivory guest and the Macintosh host. Click on the window's close box to return to the control panel.

[Help], which brings up a scrollable window containing help information. Click on the window's close box to return to the control panel.

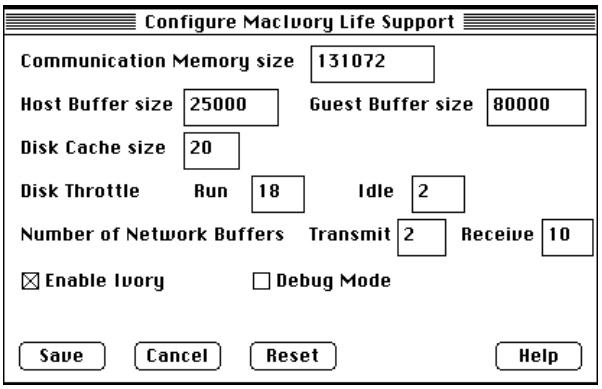

Figure 93. The Configure MacIvory Life Support Dialog

MacIvory disk partitions

Use this pane to examine and change the MacIvory disk partitions. The MacIvory control panel Help facility provides detailed information, including how to select, create, and delete a partition.

You can access the Disk Copy/Compare utility from the [Copy] button in this window. This utility can copy one disk to another or compare two disks and report any differences. The utility operates on MacIvory disk partitions or on Macintosh disks.

Initially, it is set for MacIvory disk partitions. Figure ! displays this dialog window.

Click on the [Help] button for further information on how to use Disk Copy/Compare.

See the section "Setting up a Disk for Use with MacIvory" for information on using the control panel to create Ivory disk partitions.

Using the Symbolics Keyboard with Native Macintosh Applications

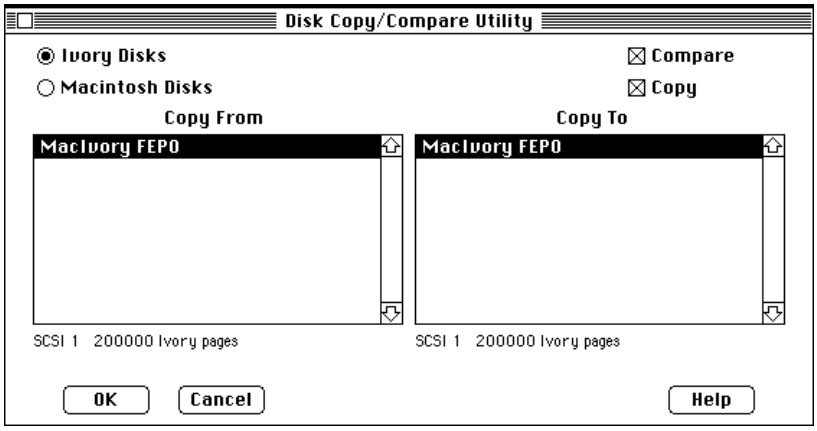

Figure 94. The MacIvory Disk Copy/Compare Utility Dialog

Use of the Symbolics keyboard with the MacIvory is provided primarily for compatibility with Genera applications. In addition, Symbolics supports the use of the Symbolics keyboard with native Macintosh applications.

Mapping of Apple Key Functions to the Symbolics Keyboard

- All "ordinary" characters (printing graphics, TAB, SPACE, RETURN) work normally.
- Apple's ESC key is entered using our ESCAPE key.
- Apple's COMMAND key is entered using either of our SUPER keys.
- Apple's OPTION key is entered using either of our META keys.
- Apple's CONTROL key is entered using either of our CONTROL keys.
- Apple's ENTER key is entered using our SCROLL key.
- Apple's CLEAR key is entered using our CLEAR INPUT key.
- Apple's HELP key is entered using our HELP key.
- Apple's END key is entered using our END key.
- Function keys (F1 - F15) are entered using the FUNCTION key as a shift key as follows:

Apple	*Symbolics*	*Apple*	*Symbolics*	*Apple*	*Symbolics*
F1	FUNCTION-1	F6	FUNCTION-6	F11	FUNCTION--
F2	FUNCTION-2	F7	FUNCTION-7	F12	FUNCTION-=
F3	FUNCTION-3	F8	FUNCTION-8	F13	FUNCTION-'
F4	FUNCTION-4	F9	FUNCTION-9	F14	FUNCTION-\
F5	FUNCTION-5	F10	FUNCTION-0	F15	FUNCTION-|

- The numeric keypad keys are entered using the SYMBOL key as a shift key as follows:

Apple	*Symbolics*	*Apple*	*Symbolics*	*Apple*	*Symbolics*
0	SYMBOL-0	6	SYMBOL-6	*	*see note.*
1	SYMBOL-1	7	SYMBOL-7	-	SYMBOL--
2	SYMBOL-2	8	*see note*		*see note*
3	SYMBOL-3	9	SYMBOL-9	.	SYMBOL-.
4	SYMBOL-4	=	*see note*		
5	SYMBOL-5	/	SYMBOL-/		

Note: In the table above, no mappings are provided for numeric 8, numeric =, numeric *, and numeric +. These four characters require special treatment because * is SHIFT 8 and + is SHIFT =. Symbolics distinguishes NUMERIC SHIFT 8 from NUMERIC * by enabling you to use the SHIFT key on the same side as the SYMBOL for producing the unshifted key with a SHIFT modifier, and using the SHIFT and SYMBOL keys on opposite sides for producing the shifted key without a modifier. Note that both SHIFT keys are required to produce a shifted key with a SHIFT modifier. For example:

```
LEFT SYMBOL 8 produces NUMERIC 8
LEFT SYMBOL LEFT SHIFT 8 produces NUMERIC SHIFT 8
LEFT SYMBOL RIGHT SHIFT 8 produces NUMERIC *
LEFT SYMBOL LEFT SHIFT RIGHT SHIFT 8 produces NUMERIC SHIFT *
LEFT SYMBOL = produces NUMERIC =
LEFT SYMBOL LEFT SHIFT = produces NUMERIC SHIFT =
LEFT SYMBOL RIGHT SHIFT = produces NUMERIC +
LEFT SYMBOL LEFT SHIFT RIGHT SHIFT = produces NUMERIC SHIFT +
```

The remaining keys on the extended keyboard are entered using the SYMBOL key as a shift key as follows:

Apple	*Symbolics*	*Apple*	*Symbolics*
↑	SYMBOL-i	HOME	SYMBOL-k
↓	SYMBOL-,	PAGE UP	SYMBOL-PAGE
←	SYMBOL-j	PAGE DOWN	PAGE
→	SYMBOL-l	DEL FWD	SYMBOL-RUBOUT

Interoperability of the Symbolics Keyboard and Popular Macintosh Software

Certain popular Macintosh software does not interoperate fully with the Symbolics keyboard. In some cases, a simple workaround (for example, renaming a file) exists which enables full function. In other cases, there is no workaround but the limitations are well known and are presented here for your convenience.

The Symbolics Keyboard and CloseView

CloseView, from Apple, allows the visually handicapped to magnify portions of their screen to make reading the screen easier. When using a Symbolics keyboard, however, the keystrokes used to raise and lower the magnification factor, Command-Option-↑ and Command-Option-↓, respectively, can not be entered using the techniques described in "Mapping of Apple Key Functions to the Symbolics Keyboard". Instead, you must disable magnification using Command-Option-X, open the CloseView control panel, change the magnification factor by clicking on the arrow buttons, close the CloseView control panel, and re-enable magnification using Command-Option-X.

The Symbolics Keyboard and Pyro!

Pyro!, from Fifth Generation Systems, is one of the most popular screen saver facilities for the Macintosh. Once running, Pyro! waits for you to press any key or move the mouse as an indication that you wish to restore the normal screen contents. However, when using a Symbolics keyboard, Pyro! will not deactivate itself if you press any of the righthand modifier keys (that is, SHIFT, CONTROL, META, and SUPER). Just use the lefhand modifier keys instead and Pyro! will deactivate.

The Symbolics Keyboard and QuickKeys 2

QuickKeys 2, from CE Software, is one of the most popular keyboard macro facilities for the Macintosh. For proper operation, the Symbolics keyboard software must load after MacIvory's support software but before QuickKeys 2. If you have INIT-Picker, or similar software for controlling the order in which INITs are loaded, see the documentation on said software for details on how to arrange the proper loading order. If you do not have such software, you can use the fact that the Macintosh System Software loads INITs in alphabetical order to get the desired effect. In particular, you can rename either the Symbolics Keyboard INIT or the QuickKeys 2 INIT to obtain the proper loading order. (We recommend that you rename Symbolics Keyboard to MSymbolics Keyboard.)

Known Keyboard Problems in Using the Genera Application with MacIvory

This section describes some keyboard problems that occur with both the Symbolics keyboard and the Apple extended keyboard when using the Genera application.

- The software flag that enables the posting of key up events to the application was not designed with MultiFinder in mind.

If the ADB bus interrupt for the button release goes off while another application or part of the system is running, no key up event is posted to the application that requested them. Apple has acknowledged this problem in a recent technical memorandum. Genera tries to work around this by polling the system's internal keys state looking for lost transitions. This polling results in some small performance degradation, but is mostly effective. Synchronization anomalies are rare but still possible.

- The event queue is very small.

  If the Macintosh is very busy processing high-priority disk interrupts, it may not run the application often enough to read out pending keyboard events and pass them on to the Ivory. If keyboard hardware interrupts occur while the event queue is full, the transitions are lost. Since the Macintosh does not normally operate the disk and network as hard asynchronously, and many applications do not support typeahead consistently, this is not a major problem for native applications. In any case, typeahead is not as reliable on the MacIvory as on other Symbolics machines.

  The size of the event queue is determined by a field in the disk's boot block. Apple does not distribute an application for modifying this. Additionally, it has forbidden the distribution of licensed application software that does so.

Configuring the Logitech MouseMan for Use with a MacIvory

You must be running MacIvory Support Software Version 4.2 to use the Logitech MouseMan with MacIvory. These instructions presume that you have already installed the MouseMan and its software according to Logitech's documentation.

1. Open the Mouse Key control panel.

2. Click on the Add button and select your copy of the Genera application.

3. Change the assignments for the middle and right mouse buttons to be Click.

   Repeat steps 2 and 3 for the Unassigned application as well.

4. Close the Control Panel.

You can now use all three buttons on the MouseMan just as you would use the buttons on a Symbolics mouse while in Genera. The MouseMan will continue to behave according to your previous configuration instructions outside the Genera application.

Running Two Genera Releases on a Single MacIvory

It is possible, although tricky, to allow two different major releases to coexist on the same MacIvory system. This section describes how to prepare for running two Genera releases (what you need to keep, and how to organize the files), and how to switch from one release to the other.

For the purpose of a specific example, we assume you are running the Genera 7.4 and Genera 8.1 releases.

Preparation for Running Two Genera Releases

For each Genera release, you need to keep the Genera world, the corresponding FEP (kernel and flod files), and all Macintosh software (the three folders: MacIvory Applications, MacIvory System, and MacIvory Development).

For safety's sake, you should keep the floppies or CD-ROM distributed from Symbolics in a safe place, in case any of these files are deleted accidentally.

You should rename the folders so that they clearly represent the software in them:

- The Genera 7.4 versions of the folders should be named: 7.4 MacIvory Applications, 7.4 MacIvory System, and 7.4 MacIvory Development.

- The Genera 8.1 versions of the folders should be named: 8.1 MacIvory Applications, 8.1 MacIvory System, and 8.1 MacIvory Development.

You need to have three sets of HELLO.BOOT and BOOT.BOOT files, one for Genera 7.4, one for Genera 8.1, and one to identify which release to run:

- The 7-4-HELLO.BOOT file should load the FEP version for Genera 7.4. The 7-4-BOOT.BOOT file should load the Genera 7.4 world.

- The 8-1-HELLO.BOOT file should load the FEP version for Genera 8.1. The 8-1-BOOT.BOOT file should load the Genera 8.1 world.

- The HELLO.BOOT file should contain "Hello 8-1" or "Hello 7-4"; this indicates which of the two above HELLO.BOOT files to use. The BOOT.BOOT file should contain "Boot 8-1" or "Boot 7-4"; this indicates which of the two above BOOT.BOOT files to use.

Switching From One Genera Release to the Other

Assume you are running Genera 8.1 and want to switch to running Genera 7.4. Follow these steps:

1. *Before shutting down Genera 8.1*, edit the HELLO.BOOT file to contain "Hello 7-4". Edit the BOOT.BOOT file to contain "Boot 7-4".

2. *Before shutting down Genera 8.1*, you need to switch FEP kernels. When you are running Genera 8.1, you can do this by using the Edit Disk Label command. When switching from Genera 8.1 to Genera 7.4, you would indicate the I311 FEP (which was the 7.4 FEP).

Note, however, when you are running Genera 7.4, the Edit Disk Label command is not defined. Thus, to switch from Genera 7.4 to Genera 8.1, you need to use **si:install-fep-kernel**. For example, to switch to the I325 FEP which is the 8.1 FEP, you would evaluate the following form:

```
(si:install-fep-kernel "I325-kernel")
```

3. Shutdown Lisp, by choosing "Shut Down" from the Ivory pulldown menu. This quits the Genera application.

4. Copy the contents of the 7.4 MacIvory System folder to the System folder. **Be sure to hold down the Option key while dragging to copy, not move the contents of the folder**; you should always keep the contents of the 7.4 MacIvory System folder so you can copy it again later. The system asks if you want to replace the files with the same names; you should answer Yes.

5. Restart the Macintosh.

6. When the Macintosh is running again, open the 7.4 MacIvory Applications folder and run the Genera application. If you are asked whether you want to cold boot, answer Yes. (If you are not asked, then the system cold booted automatically.)

To switch from Genera 7.4 to Genera 8.1, use the same process, but switch "8-1" and "7-4".

Programmer's Reference to MacIvory

Remote Procedure Call for the Macintosh

This section presents information on using the Symbolics implementation of Remote Procedure Call (RPC) with MacIvory applications. It includes an overview of Symbolics RPC and a description RPC facilities specific to the MacIvory.

Overview of Symbolics RPC

Symbolics RPC is an implementation of industry-standard RPC that underlies Sun Microsystems' NFS and other programs (see Request for Comments (RFC) #1057 "RPC: Remote Procedure Call Protocol specification version 2"). The distinguishing characteristic of Symbolics RPC is that it uses Lisp technology to provide a very clean and easy-to-use interface for defining RPC-based programs. The form of data transmitted over the communications medium is fully compliant with the standard.

Remote Procedure Call (RPC) is a facility that allows a function executing on one processor to call a function executing on another processor. The two functions can be written in the same language or in different languages, such as Lisp and C. The two processors can be of the same type or of different types; for example, a function executing on an Ivory can call a function that executes on an MC68020.

RPC allows a program executing on one processor to access facilities that are available on another processor. For example, an Ivory embedded in a host can use RPC to make use of hardware devices controlled by that host, to call facilities of the host operating system, and to call program libraries that are available for the host but not for the Ivory. Similarly, a program running on a host can use RPC to call symbolic processing facilities such as Joshua that run on the Ivory.

Using RPC, you can segment a program into pieces and run each piece on a different processor. This can improve performance through parallel processing. More importantly, this allows each part of the program to execute on the processor and under the operating system best adapted to support that part. Benefits include both performance improvement and ease of programming.

For example, a program for a MacIvory system can run its user interface on the Macintosh and its knowledge processing on the Ivory. It is not necessary to have such a large granularity in the segmentation of a program; the same program might be improved by running the high-level "policy" portion of its user interface on the Ivory, with the low-level "mechanism" portion running on the Macintosh. Dynamic Windows on MacIvory work precisely this way.

Another reason to use RPC is when you want to run a program on processor A but it needs to cooperate with an existing program that is available only on processor B. Processor A might be an Ivory, which you are using because of its ease of programming, while processor B might be a non-Symbolics processor, with a large library of available programs. The main part of your program runs on processor A and it includes an appendage that runs on processor B; the appendage communicates with the existing program using the interfaces defined by the existing program. The main part of your program and the appendage communicate through RPC. The existing program is unaware of RPC and does not have to be modified or adapted. (The Genera interface to HyperCard on MacIvory works this way.)

RPC provides communication between two processors in a single system, as when a Symbolics Ivory is embedded in a non-Symbolics platform such as a Macintosh or Sun.

In this case communication is through shared memory and is quite efficient, although of course calling a function remotely is never as fast as calling it locally. RPC can also be used for communication between two processors in separate systems, which might be physically located side by side or at a great distance from each other. RPC operates through local-area and wide-area networks and through RS232 serial lines. Using RPC over a network is slower than using RPC in an embedded system.

Symbolics RPC provides a transparent interface; calling a function remotely looks the same as calling a local function. When you call a function, you do not have to know whether its body executes on the local processor or on a remote processor. This is true regardless of whether you program in Lisp or in C. The RPC system implements this by automatically defining a *stub function* that acts as a local representative of the remote function. The stub takes care of all the housekeeping required to transmit the arguments to the remote function and receive back the values. Symbolics RPC provides a transparent interface for the callee as well. You

write the body of a remotely callable function in Lisp or C in the usual way; the RPC system automatically adds code to receive the arguments, puts them in variables with the names you specified, and sends back the results.

Symbolics RPC and Sun RPC

Symbolics RPC is a fully compliant implementation of the RPC and XDR *(eXternal Data Representation)* standards described in RFC (Request for Comments) #1057 "RPC: Remote Procedure Call Protocol specification version 2" and RFC #1014 "XDR: External Data Representation standard."

As such, it is completely compatible and can interoperate with any other compliant RPC implementation, such as the one supplied with Sun Microsystems computers. (See the Sun Microsystems document *Network Programming* for further information.) For instance, a program written in Symbolics RPC can make RPC calls to a program written in SunRPC language, and vice versa.

Symbolics RPC language differs from SunRPC language in many ways, most notably in that Symbolics RPC can simultaneously generate code in two programming languages, C and Lisp. Symbolics RPC language cannot generate code in Sun RPC language. Users of the Symbolics UX can choose either for programming. Symbolics RPC language is likely to make the code-maintenance task easier for programs that will run on both Ivory-based systems and a C-based system.

Differences Between Local and Remote Function Calling

An important and necessary difference between local and remote function calling is that functions executing on separate processors have separate memory address spaces and cannot share any data. All arguments and values must be passed by value, not by reference. For this reason, unlike a locally callable function, a remotely callable function uses special functions (**rpc:rpc-values** and **rpc:rpc-error** in Lisp, RPCValues and RPCError in C) to return its results.

Because there is no call by reference, the data types that can be used with RPC are limited. For example, in Lisp you cannot pass an arbitrary symbol as an argument. If you pass a flavor instance, the callee sees a copy of the instance. If the callee modifies the instance, those modifications are not passed back to the caller. On the other hand, a benefit of call by value is that the caller and callee can use different data representations. For example, the caller can pass a Lisp flavor instance, which the callee will see as a C struct.

You construct an RPC-based program by using a set of Lisp macros to define the remotely callable functions. These Lisp macros are somewhat unusual in that they expand into both Lisp code and C code. The Lisp expansion is processed in the normal way. The C expansion is written to a file that can be compiled by the Symbolics C compiler or shipped to another processor and compiled by its own C compiler. Once the interface has been defined and compiled, you call the stub functions using ordinary Lisp or C function calls. The callee or server half of the interface is loaded together with any other programs it calls.

Basic Concepts of RPC

The basic concepts of RPC include *remote modules, remote entries, remote errors*, and *remote types*, explained in the following table:

remote entry	A remotely callable function.
remote module	A collection of related remote entries that are treated as a unit for bookkeeping purposes.
remote error	An exceptional condition that can arise while executing a remote entry.
remote type	A type of data that can be used as an argument to a remote entry or a remote error, and can be returned as a value by a remote entry. A remote type defines the possible data values, their representation in Lisp and C, their representation for interprocessor transmission, and the methods for converting between these representations.

Some of these concepts have nonstandard names. These names were chosen to avoid any confusion with other concepts in Genera with names similar to the standard names. Other systems call remote modules "remote programs" and call remote entries "remote procedures."

The RPC facility consists of three layers:

- The *call layer* is in charge of identifying remote entries to be called, transmitting the arguments to them, matching up the returned values with the caller who is awaiting the results, and reporting errors.

- The *data representation* layer is in charge of defining a common representation for data and translating representations used by different machines and by different programming languages to and from the common representation.

- The *transport layer* is in charge of moving raw bits between machines and dealing with bit-ordering issues. There are three different transport layers, selectable at run time. One is based on the embedding substrate's inter-processor communication mechanism, the others are based on the byte-stream and UDP/IP media of the generic network system.

Types of RPC Servers for MacIvory

The **:type** option to **rpc:define-remote-c-program** defines three types of servers: linked, auto-load, and auto-load-with-static-data. When choosing which type of server to use, your first decision is whether to use a linked server or an auto-load server. A *linked server* is an integral part of an application program, built into that program and able to call subroutines of it and access its static data. An *auto-load server* is an independently created module that executes inside an application program without any knowledge of the structure of the application.

If you are not writing your own application program, your choice is simple, because only auto-load servers are possible; to use a linked server you must have a program to link it into. If you are writing your own application program, whether to use a linked server or an auto-load server depends mainly on whether you want to maintain the server independently, or tie it closely to the application program.

If you are running the Symbolics-supplied application program, such as the Genera icon or other icons that you created from the Symbolics-supplied Genera or Unassigned icon using the Configure MacIvory Application command, you must use an auto-load server, since you cannot modify this program to link in your own servers. The same auto-load server can also be used with your own application programs. The Macintosh Toolbox interface is implemented entirely with auto-load servers so that it is available in all applications, including the ones you write yourself.

On the Macintosh, an auto-load server only occupies memory if it is called, while a linked server may or may not occupy memory even if it is not called, depending on whether you direct the linker to put it in its own segment.

If you choose an auto-load server, your next decision is whether to use **:auto-load-with-static-data** or plain **:auto-load**. In general, the **:auto-load** type is preferable because it allows you to set the "purgeable" bit in the resource, which in turn allows your server to be swapped out when it is not executing if the Macintosh gets low on memory. However, you must use **:auto-load-with-static-data** if you have static variables in your C code or use the **:init** option to **rpc:define-remote-c-program**.

The "purgeable" bit in each RPCD resource controls memory allocation for that code segment. You can use the Attrs item in [Project / Set Project Type...] in THINK C to set this bit before compiling your program. If the "purgeable" bit is off, the code segment is loaded in to memory and locked when your remote module is first called and remains in memory at a fixed address forever (until the application quits). This is the normal setting. If the purgeable bit is on, the code segment can be moved to a different memory address, or unloaded from memory entirely, any time a call to it is not actively in progress. Set the purgeable bit if you want to minimize memory consumption, but only if your program does not use interrupts, VBL tasks, call backs, or anything else that could fail if the program's memory address changes.

An auto-load server is always locked while it is executing. If you do not set the purgeable bit, it is locked at all times, but if you do set the purgeable bit, it is unlocked between calls. You should not set the resource's "locked" bit nor its "sysHeap" bit. The purgeable bit should not be used for **:auto-load-with-static-data** RPC servers. If you use it, the values of your static variables will be reset at unpredictable times.

The Macintosh has certain limitations on code resources. Since auto-load and auto-load-with-static-data RPC servers are implemented as code resources, they are subject to these limitations. The maximum size of a code resource is 32K bytes. If you exceed this limit you may be able to work around it by breaking up your **rpc:define-remote-module** into several separate modules. Code resources access their global variables through register A4 instead of the normal register A5. The

RPC system maintains these registers for you, but problems can still occur if you use libraries. You may need to copy and recompile any libraries that you use, as explained on page 85 of the THINK C manual. If you access certain QuickDraw globals such as the built-in cursor shapes and stipple patterns you will find that they don't work correctly.

Auto-loading Servers for RPC

This section tells you how to create an auto-loading RPC server. It provides a sample remote program and a procedure for creating an auto-loading server. The server is loaded into an RPC-bearing Macintosh application, such as "Genera", the first time the Ivory tries to call it.

First, define your remote program. You can find the one used in this example, called remote-program-example.lisp in the file SYS:EMBEDDING;RPC;EXAMPLES;REMOTE-PROGRAM-EXAMPLE.LISP.

Remote Program Example

```
;;; -*- Mode: LISP; Syntax: Lisp+C; Package: USER; Base: 10 -*-
;;;> EXAMPLES-MESSAGE
;;;>
;;;>*****************************************************************************************
;;;>
;;;>       Symbolics hereby grants permission to customer to incorporate
;;;>       the examples in this file in any work belonging to customer.
;;;>
;;;>*****************************************************************************************

;;; A simple example of how to use RPC on MacIvory
;;; This uses the Mac Standard File package to read a file name

;;; This could be done through the toolbox interface,
;;; but for purposes of this example, we are not
;;; using the interface, instead doing the work by hand.

;; Assign a number to the remote module and declare that it will
;; be used for Lisp-to-C calls
(RPC:DEFINE-REMOTE-MODULE EXAMPLE (:NUMBER #x7F008000) (:VERSION 1)
                                  (:CLIENT :LISP) (:SERVER :C))

;;; Define the types that we will need

;; Macintosh points.  The Lisp representation is just (VECTOR V H).
(RPC:DEFINE-REMOTE-TYPE POINT ()
  (:ABBREVIATION-FOR '(RPC:STRUCTURE (V RPC:INTEGER-16) (H RPC:INTEGER-16)
                       (:C #{ Point }))))
```

```
;; List of what file types we will accept, or wildcard that accepts all file types
(RPC:DEFINE-REMOTE-TYPE FILE-TYPES ()
  (:ABBREVIATION-FOR '(OR ALL (VECTOR FILE-TYPE))))

(RPC:DEFINE-REMOTE-TYPE ALL ()
  (:ABBREVIATION-FOR '(MEMBER :ALL)))

;; Macintosh file-type codes.  The Lisp representation is a 4-character string.
;; Since in C these are treated as long integers rather than arrays, we need
;; to write some C code sending and receiving them instead of relying on
;; the default action for vectors
(RPC:DEFINE-REMOTE-TYPE FILE-TYPE ()
  (:ABBREVIATION-FOR '(VECTOR RPC:CHARACTER-8 4))
  (:C
    (:DECLARE (NAME) #{ OSType ↓NAME })
    (:SEND (VALUE)
      #{ send_word( ↓VALUE ); }
      )
    (:RECEIVE (VARIABLE STORAGE-MODE)
      (PROGN (IGNORE STORAGE-MODE)
             #{ ↓VARIABLE = receive_word(); }
             ))))

;; Define the RPC interface to the Macintosh SFGetFile routine
;; We tell it where to put the dialogue box on the screen and what
;; file types we are interested in.  It interacts with the user and returns
;; whether the user confirmed or cancelled, the volume (really directory)
;; identifying number, the file name, and the Macintosh file type code).
(RPC:DEFINE-REMOTE-ENTRY READ-FILE-NAME EXAMPLE (:NUMBER 1)
                         (:ARGUMENTS (WHERE POINT)
                                     (FILE-TYPES FILE-TYPES))
                         (:VALUES (CONFIRMED RPC:BOOLEAN)
                                  (VOLUME RPC:INTEGER-16)
                                  (FILE-NAME RPC:PASCAL-STRING)
                                  (FILE-TYPE FILE-TYPE))
  ;; Define the C glue code needed to interface to the operating system
  ;; Put the list of file types into the form required by SFGetFile
  ;; and extract the values from the SFReply structure
  (:C
    (:SERVER
      #{ SFReply reply;
         int numTypes;
```

```
        if ( file_types.type == all_type )
          numTypes = -1;
        else numTypes = file_types.value.vector->length;
        SFGetFile(where, "\p", NULL, numTypes, &file_types.value.vector->element[0],
                  NULL, &reply);
        RPCValues(reply.good, reply.vRefNum, reply.fName, reply.fType);
      }}))

;; Output the C code to a separate file, for compilation on the Macintosh
;; We only need server code
(RPC:DEFINE-REMOTE-C-PROGRAM EXAMPLE
  (:SERVER
    (:FILE "example.c")
    (:TYPE :AUTO-LOAD)
    (:INCLUDE "<MacTypes.h>" "<FileMgr.h>" "<StdFilePkg.h>")))
```

Notes:

1. Note that `Syntax` in the attribute line is set to `Lisp+C`. This enables the `#{...}` syntax.

2. In interprocessor communication, modules are identified by their number, not by name. Choose module numbers following the conventions in the file SYS:EMBEDDING;RPC;ASSIGNED-NUMBERS.TEXT . All module numbers for a particular site must be unique.

3. You can replace the type **:auto-load** with **:auto-load-with-static-data** if you have static variables in your C code or use the **:init** option to **rpc:define-remote-c-program**. In general, the **:auto-load** type is preferable to **:auto-load-with-static-data** because it allows you to set the "purgeable" bit in the resource, which in turn allows your server to be swapped out if the Macintosh gets low on memory.

4. The **:include** files are whatever include files your remote entries need. Here we need general Macintosh types such as Point, the Macintosh File Manager, and the Standard File package.

Prerequisites

Before beginning this procedure:

- You must have THINK C loaded on your Macintosh disk.

- From the Macintosh Finder, create a folder called RPC-Example on your Macintosh disk. This folder should not be in any other folder.

The procedure is divided into these steps:

1. Compiling the example program in Genera
2. Copying the C program to the Macintosh
3. Compiling and linking the C program on the Macintosh
4. Installing the code resource file
5. Calling the remote module

Procedure

1. **Compile the example program in Genera.**

   a. Switch to Genera by double-clicking on the Genera application icon.

   b. Compile and load the example program from the Lisp Listener with the Compile File and Load File commands.

   This tells Genera about the remote module, and produces a C program in the file SYS:EMBEDDING;RPC;EXAMPLES;EXAMPLE.C specified in **rpc:define-remote-c-program** (line 77).

   The next step is to use the example.c file to tell the Macintosh about the remote module.

2. **Copy the C program to the Macintosh.**

   Use the Copy File command to copy the C program example.c from Genera to the RPC-Example folder on the Macintosh.

   For example,

   ```
   Copy File sys:embedding;rpc;examples;example.c  HOST:disk:RPC-Example:example.c
   ```

   In this example, HOST is used literally to specify the local Macintosh, *disk* is the name of that Macintosh's hard disk (the name displayed under the disk icon), and RPC-Example is the name of the folder. **Note**: If the disk has a space in its name, you must enclose the entire pathname in double quotes.

3. **Compile and link the C program on the Macintosh.**

   This step describes the work necessary to compile and link the C program on the Macintosh. It tells you how to set up a THINK C project and how to compile and link your C program with it. See THINK C documentation for further information.

Before compiling the C program, create a file named RPC.h and a file named MacIvory-Suypport.h in the RPC-Example folder. The file should contain a #include of the right file name (for example, "::MacIvory Development:RPC:RPC.h"). Follow these steps for this example:

a. From Genera, type:

```
Edit File HOST:disk:RPC-Example:rpc.h
```

b. This places you at an editor window. Type in the following text:

```
#include "::MacIvory Development:RPC:RPC.h"
```

c. Save the buffer by typing `c-X c-S`.

d. Follow a similar procedure to create a file named MacIvory-Support.h. Type the following text in the file:

```
#include "::MacIvory Development:Substrate:MacIvory-Support.h"
```

The next task is to compile your C program and link it with the libraries or other C source files called by the C code in the **:server** clauses of your **rpc:define-remote-entry** forms. In this example, we use the THINK C compiler. If you are unfamiliar with this product, read the THINK C manual for background.

This example uses one library, one source file from the RPC system, and no additional source files. (**Note**: For large C programs, which are not just calls to the Macintosh operating system, you might put the bulk of it in a separate file and put function calls in the **rpc:define-remote-entry** forms. This makes editing easier because you can edit the C code in C mode instead of Lisp mode.)

a. Switch from Genera to the Macintosh Finder and start up THINK C. You are presented with a THINK C file selection box.

b. From this box, move to the RPC-Example folder on the disk. (see "Starting THINK C" in the THINK C manual). To do this:

   i. Click on the THINK C open folder icon and move to *disk*, where *disk* is the name of your disk.

   ii. From the directory list of the contents of your disk, double-click on RPC-Example to open it.

c. Create a THINK C project called `example proj` in the RPC-Example folder

The THINK C compiler has a number of options. This example assumes a particular setting of these options, which is normally the default, unless the default has been changed at a particular installation.

In [Edit / Options...], there are 5 sub-menus selected by radio buttons, of which we care only about two. In Code Generation, Macsbug Symbols and <MacHeaders> are assumed to be on, and the others are assumed to be off. In Compiler Flags, Check Pointer Types is assumed to be on, and Require Prototypes must be off.

After setting the options, if necessary, create a project with these steps:

i. Click on New in the file selection box.

ii. At the prompt, Name New Project, type: `example proj`.

d. Using [Source / Add ...] while still running THINK C, add the following three files to example proj:

(Continue to work from the file selection box as described in the previous step.)

- Add the example.c file from the RPC-example folder on the disk.
- Add the file rpcspprt.c in the RPC folder in the MacIvory Development folder.
- Add the MacTraps library in the Mac Libraries folder in the THINK C folder.

e. Click Cancel after adding those three files, to get out of the file menu.

Set the project type and related information. Select [Project / Set Project Type] from the pull-down menu and set the radio button to Code Resource. Specify the type (in the box) as RPCD (you must use all caps), the ID as 1, and the name to whatever you want (leaving it blank is okay). Be sure to type RPCD in all capital letters. Click OK and then OK again. This procedure tells THINK C to build a code resource file.

Note: The "purgeable" bit in each RPCD resource controls memory allocation for that code segment. You can use the Attrs item in [Project / Set Project Type...] in THINK C to set this bit before compiling your program. If the "purgeable" bit is off, the code segment is loaded in to memory and locked when your remote module is first called and remains in memory at a fixed address forever (until the application quits). This is the normal setting. If the "purgeable" bit is on, the code segment can be moved to a different memory address, or unloaded from memory entirely, any time a call to it is not actively in progress. Set the "purgeable" bit if

you want to minimize memory consumption, but only if your program does not use interrupts, VBL tasks, call backs, or anything else that could fail if the program's memory address changes.

f. Use [Project / Build Code Resource] to compile, link, and produce the output, with whatever name you want.

g. The same name as in **rpc:define-remote-module** is customary, so for this example you put the output in a Macintosh file named example in the RPC-Example folder. Note: You will be in Mac Libraries and need to switch to RPC-example folder. Now you can use [File / Quit] to leave THINK C.

4. **Install the code resource file.**

The next step is to make the RPC system aware of the code resource file you just built. You need to do two Macintosh-oriented things:

- Set the file's type and creator attributes to RPCS and IVRY instead of the default.

- Create a resource of type RPCS with ID 1 (same ID as your RPCD resource).

This resource should be 8 bytes long, where the first 4 bytes are the **:number** option from **rpc:define-remote-module** and the second 4 bytes are the **:version** option. Remember about decimal versus hexadecimal.

If you know how to use ResEdit, you can do these operations graphically in ResEdit. Alternatively, you can do them textually in RMaker as described in the following procedure.

a. Use the Genera command Copy File to copy the file SYS:EMBEDDING;RPC;EXAMPLES;EXAMPLE.R into the RPC-Example folder. Type:

```
Copy File sys:embedding;rpc;examples;example.r Host:Disk:RPC-Example:example.r
```

(**Caution**: RMaker requires that the text in this file appear exactly as given, including spaces and blank lines. This includes the blank line that ends the file.)

b. Run RMaker and select the example.r file from the file menu. (You can use the Find File desk accessory, if you need help in finding RMaker.) When it finishes, click on the Quit button. You now have an Example Server file.

c. Move this file into a folder where the RPC system can find it, which can be the folder where "Genera" is launched from, or the folder *Disk*:System

Folder: Ivory:RPC: (the latter is preferable, since it keeps RPC servers separate from everything else).

Now calling the remote module from Genera should work.

5. **Call the remote module.**

To call the example module, switch back into Genera and evaluate the form:

```
(read-file-name #(150 100) :all)
```

A standard Macintosh file menu appears near the middle of the screen. First try clicking the Cancel button. You get back values of NIL and three garbage values. Try it again and select a file, and you will get T, a negative number that encodes what folder the file is in, the name of the file as a string, and the Macintosh type of the file as another string.

Incorporating RPC Into Macintosh Applications

Symbolics provides a THINK C version 4.0 library named RPC.lib that implements the Remote Procedure Call protocol between the host Macintosh and its embedded Ivory. The Genera application provided by Symbolics uses this library.

You can link the RPC library into your own Macintosh applications. This allows you to use RPC to request services from the Ivory in your application by linking in stub files written by the **:client** option of **rpc:define-remote-c-program**.

In addition, the RPC library makes your application an RPC server for requests from the Ivory. The RPC library transparently provides all the support for auto-load RPC servers (see the section "Auto-loading Servers for RPC" for further information). You can also link RPC server files written by the **:server** option of **rpc:define-remote-c-program** into your application. Programs running on Ivory can then call those servers.

In order to use the RPC.lib library, you must also link in the MacIvory library provided by Symbolics.

The MacIvory library provides facilities for booting the Ivory, displaying the cold load stream, and related tasks. For an example of the use of these facilities, see the example program in the file SYS:EMBEDDING;RPC;EXAMPLES;APPLICATION-EXAMPLE.C.

The RPC library is available on the Macintosh in the file *disk*:MacIvory Development:Libraries:RPC.lib, where *disk* is the name of your Macintosh's hard disk. The MacIvory library is available on the Macintosh in the file *disk*:MacIvory Development:Libraries:MacIvory.lib, where *disk* is the name of your Macintosh's hard disk.

In addition to these libraries, Symbolics also supplies the libraries RPC A4.lib and MacIvory A4.lib. These are identical to the MacIvory and RPC libraries except that they have been compiled to use register A4 instead of A5. Use these libraries when you are building code resources rather than applications. (See the section "Using libraries in code resources" in the *THINK C User's Manual* for further information.)

Your Macintosh application must be written in THINK C in order to use the RPC library.

Initializing the RPC and MacIvory Libraries

Your Macintosh application must initialize the RPC and MacIvory libraries before using them. Follow these steps to initialize the libraries:

1. Initialize the Macintosh operating system and any other facilities you use.

2. Call the functions `InitMacIvorySupport` and `InitMacIvory`.

3. Call the function `InitializeRPC` with no arguments. It does not return anything.

4. Call `initialize_`*`remote_module_name`*`_server` with no arguments, for every RPC server file that you have linked into your application. You must always call `initialize_predefined_remote_entries_server`, since that RPC server is included in the RPC library. These functions return a standard Macintosh error code of type OSErr; however, it is generally safe to assume that no error occurred.

5. Call the function `emb_agent_open` with no arguments. It returns a standard Macintosh error code and establishes a bidirectional channel between the host Macintosh and the embedded guest Ivory. If the value is not noErr, you should terminate the application. It is important to check this error code. If `emb_agent_open` fails, call the `ReportRPCOpenFailure` routine to issue a standard Macintosh alert box.

Terminating the RPC and MacIvory Libraries

Your Macintosh application must terminate the RPC and MacIvory libraries before it exits. Follow these steps:

1. Optionally, call `OKtoStopMacIvory`. (Use this to issue the standard "Lisp is running. Quit?" alert box). This function returns a value of TRUE or FALSE.

2. Call the function `CloseRPC` with no arguments. This function does not return anything. If you exit without calling `CloseRPC`, the interprocessor communication channel allocated by `emb_agent_open` remains permanently busy.

3. Call `TermMacIvorySupport`.

4. Call the function `ExitToShell`.

Note: You must provide a function, `ExitMacIvoryApplication`, which is called by the MacIvory library if it decides it is impossible to continue. This function must perform the last three actions above. Therefore, it is possible to replace steps 2 through 4 with a call to your `ExitMacIvoryApplication` function.

Interfacing the RPC and MacIvory Libraries to an Event Loop

The RPC library interfaces with your application's event processing loop in two directions: you call it, and it calls you. You must call the RPC library periodically, so that incoming requests from the Ivory can be serviced. Normally, this is done from your event processing loop. Even if you do not plan to make any RPC calls from Ivory to Macintosh, you still must call the RPC library to support internal housekeeping.

The RPC library calls back to your event processing loop whenever it has to wait for some event to occur, normally a response from the Ivory. This happens when you make an RPC call from Macintosh to Ivory, for example. While the Macintosh waits for the Ivory to complete the call and return the values, the RPC library repeatedly calls your event loop.

Your event processing loop should call the function `PollRPC` with no arguments. It does not return any result. Call `PollRPC` before calling `WaitNextEvent` or `GetNextEvent` or in the same circumstances in which you call `SystemTask`.

`MacIvoryTasks` must be called periodically, typically from your application's main event loop. It handles reset requests from Ivory and maintains the cold-load-stream window. Like `PollRPC`, `MacIvoryTasks` should be called before a call to `WaitNextEvent` or `GetNextEvent`, or in the same circumstances in which you call `SystemTask`.

Your application's event processing loop should call `MacIvoryEvent` on every event, before processing it. `MacIvoryEvent` takes care of the Ivory menu and the cold-load-stream window.

Your application must define the function `BusyWait`, to be called by the RPC library when it needs to wait. `BusyWait` takes one argument, an `int` named `allow_rpc`, and returns no values. `allow_rpc` is true if the RPC system expects you to call it back, or false if it does not. BusyWait should call `MacIvoryTasks`. You should call `PollRPC` if and only if `allow_rpc` is true.

`BusyWait` gives you an opportunity to implement whatever multiprocessing strategy you prefer. When using MultiFinder, `BusyWait` must call `GetNextEvent` or `WaitNextEvent` so that other Macintosh applications can run. In general it's a good idea for `BusyWait` to support the mouse at least to the extent of allowing the Apple menu to be used, but it is not a good idea for `BusyWait` to run portions of your application that can make RPC calls to the Ivory.

You must define a routine named `RestartMacIvoryApplication`, which is called with no arguments from `MacIvoryTasks` when the Ivory system is booted or restarted. `RestartMacIvoryApplication` should reset whatever is appropriate to reset in your application. You must call `CloseRPC` in this function and then open a new channel.

You must define `ExitMacIvoryApplication`. This routine, which takes no arguments, cleans up after your application, calling `CloseRPC`, `TermMacIvorySupport` and `ExitToShell`.

You must define a function `NoteMacIvoryStateChange`. It is called by the library whenever Ivory's state changes (that is, starts up, shuts down and so on). `NoteMacIvoryStateChange` is called with one agrument of the type `enum MacIvoryStateTransisition`, which indicates what just happened to the Ivory.

Using RPC in a MacIvory Application

Once the RPC library has been initialized, you can make RPC calls to Ivory simply by calling the function. The **:client** option of **rpc:define-remote-c-program** creates a file with the necessary "stub" code that does the interprocessor communication. Link this file into your application and RPC calls look like ordinary C function calls. All RPC calls return a standard Macintosh error code, which should be checked by the caller. If the RPC call has values, you pass extra arguments which are the addresses of the variables to receive the values.

Because your application includes the RPC library, your application automatically supports dynamic loading of RPC servers defined with the **(:type :auto-load)** suboption of the **:server** option to **rpc:define-remote-c-program**. You can also link RPC servers directly into your program, which is useful when the servers share data or subroutines with the rest of your application. Use the **(:type :linked)** suboption of the **:server** option to **rpc:define-remote-c-program**, and remember to call initialize_*remote_module_name*_server.

If you issue several asynchronous RPC calls in rapid succession, you can't rely on the Ivory executing these calls in the order in which they were issued. In general the Ivory will execute all of the calls in parallel, each in a separate process. (Note: an asynchronous call is a call to an entry whose **rpc:define-remote-entry** form uses the **:asynchronous** option.)

If you depend on asynchronous RPC calls to be executed one at a time in order, you should use the **(:process nil)** option of **rpc:define-remote-module**. This causes all calls to entries in that module to be executed in the RPC Dispatch process, rather than in separate processes, which means that each call will be fully processed before handling of the next call commences. This feature should be used with caution, because if your server doesn't return, the RPC Dispatch process will be out of operation and the Ivory will appear to be dead and non-responsive.

Note that if some entries of a particular remote module should be executed in the RPC Dispatch process while other entries should be executed in separate processes, you should either put the latter entries into a separate remote module or call **process-run-function** explicitly in your server code.

Selecting the Right RPC Agent for MacIvory Applications

When multiple Macintosh applications are using the Ivory simultaneously, software that runs on the Ivory needs to know which Macintosh application it is talking to in order to perform RPC calls and have them served in the right context. If you only make Macintosh-to-Ivory calls, and not the reverse, you don't need to worry about this.

When a Lisp program makes an RPC call, it is actually calling a "stub" function that does the interprocessor communication. This stub uses the special variable **rpc:*default-transport-agent*** to tell it what server to communicate with. The global value of **rpc:*default-transport-agent*** refers to a Macintosh application such as Genera, or one created with the Configure MacIvory Application command, that provides the full range of services. When your application makes a Macintosh-

to-Ivory call, **rpc:*default-transport-agent*** is bound in the Ivory process that executes the server to the agent that connects to your application. Thus if in the process of serving a call from the Macintosh the Ivory calls back to the Macintosh, it will reach the right application.

If the Ivory side of your application uses multiple processes, you will need to pay special attention to **rpc:*default-transport-agent***. Otherwise, it is likely to have the desired value automatically.

Example of a Simple Macintosh Application

This procedure goes through the steps of building a simple example application that runs on the Macintosh and uses RPC to obtain services from the Ivory coprocessor. The example service we use is to retrieve the property list of a symbol and display it in a Macintosh window. The Plist command in the File menu does this. The usual Macintosh window manipulation commands are also provided. The source code for this example is contained in the following files:

SYS:EMBEDDING;RPC;EXAMPLES;PLIST-SERVER.LISP

SYS:EMBEDDING;RPC;EXAMPLES;APPLICATION-EXAMPLE.C

Prerequisites

Before beginning this procedure:

- You must have THINK C loaded on your Macintosh disk.
- From the Macintosh Finder, create a folder called RPC-Example on your Macintosh disk. This folder should not be in any other folder.

The procedure is divided into these steps:

- Compiling the server side of the program
- Copying the client side of the example program to the Macintosh
- Compiling the client side of the program
- Compiling and linking the C program on the Macintosh
- Running the program

Procedure

1. **Compile the server side of the example program.**

    a. Switch to Genera by double-clicking on the Genera application icon.

b. Compile and load the example server program from the Lisp Listener with the Compile File and Load File commands, operating on the file SYS:EMBEDDING;RPC;EXAMPLES;PLIST-SERVER.LISP.

This tells Genera about the remote module, and produces a C program in the file SYS:EMBEDDING;RPC;EXAMPLES;PLIST-SERVER.C specified in **rpc:define-remote-c-program**.

2. **Copy the client side of the example program to the Macintosh**

Copy the C source code to the Macintosh. Use the Copy File command to copy the stubs for the example server program to the RPC-Example folder on the Macintosh, and to copy the source code for the Macintosh application to the same folder.

For example,

```
Copy File sys:embedding;rpc;examples;plist-server.c  HOST:disk:RPC-Example:
Copy File sys:embedding;rpc;examples;application-example.c  HOST:disk:RPC-Example:
```

In this example, `HOST` is used literally to specify the local Macintosh, `disk` is the name of that Macintosh's hard disk (the name displayed under the disk icon), and `RPC-Example` is the name of the folder.

3. **Compile and link the C program on the Macintosh**.

Before compiling the C program, create files named RPC.h and MacIvory-Support.h in the RPC-Example folder. These files should contain a `#include` of the appropriate file names.

Follow these steps for this example:

a. From Genera, type:

```
Edit File HOST:disk:RPC-Example:rpc.h
```

b. This places you in an editor window. Type in the following text:

```
#include "::MacIvory Development:RPC:RPC.h"
```

c. Save the buffer by typing `c-X c-S`.

d. Type `c-X c-F`. When asked for a file name, type:

```
HOST:disk:RPC-Example:MacIvory-Support.h
```

e. Type in the following text:

```
#include "::MacIvory Development:Substrate:MacIvory-Support.h"
```

f. Save the buffer by typing `c-X c-S`.

The next task is to compile your C program and link it with the necessary libraries. In this example, we use the THINK C compiler. If you are unfamiliar with this product, read over the THINK C manual for background.

This example uses the two source files we just copied onto the Macintosh and four libraries.

a. Switch from Genera to the Macintosh Finder and open the disk if it's not already open.

b. Double-click on the THINK C Folder to open it.

c. Double-click on the THINK C icon in the THINK C folder to start up THINK C. You are presented with a THINK C file selection box.

d. From this box, move to the RPC-Example folder on the disk. (See "Starting THINK C" in the THINK C manual). To do this:

   i. Click on the THINK C open folder icon and move to DSK (your disk).

   ii. In the directory list of the contents of DSK (your disk), double-click on RPC-Example so that it is opened.

e. Create a THINK C project called `Application-example proj` in the RPC-Example folder.

   The THINK C compiler has a number of options. This example assumes a particular setting of these options, which is normally the default, unless the default has been changed at a particular installation.

   In [Edit / Options...], there are five sub-menus selected by radio buttons, of which we only care about two. In Code Generation, Macsbug Symbols and <MacHeaders> are assumed to be on, the others are assumed to be off. In Compiler Flags, Check Pointer Types is assumed to be on, Require Prototypes must be off.

   In [Project / Set Project Type...], File Type is APPL, Creator is ????, Separate STRs is checked, and MF Attrs is 0000.

   After setting the options, if necessary, create a project with these steps:

   i. Click on New in the file selection box.

ii. At the prompt, Name New Project, type: `Application-example proj`.

f. Using [Source / Add ...] while still running THINK C, add the following files to `Application-example` proj (continue to work from the file selection box as described in the previous step):

- Add the application-example.c file from the RPC-example folder on the disk.
- Add the plist-server.c file from the RPC-example folder on the disk.
- Add the RPC.lib library in the Libraries folder in the MacIvory Development folder. This library supports the Remote Procedure Call facility.
- Add the MacIvory.lib library in the Libraries folder in the MacIvory Development folder. This library supports the Ivory coprocessor.
- Add the MacTraps library in the Mac Libraries folder in the THINK C folder. This library provides the interface to the Macintosh operating system.
- Click below the dotted line in the project window to select a new code segment. (Otherwise, the application will fail to link because it is too large.)
- Add the ANSI library in the C Libraries folder in the THINK C folder. This library provides the interface to the Macintosh operating system.

g. Click Cancel after adding those files, to get out of the file menu. Optionally, select [Project / Set Project Type] from the pull-down menu and check the Separate STRs box. Click on OK. (Otherwise you will get a "can't load STRS in this project" error.)

h. Use [Project / Build Application] to compile, link, and produce the output, naming it Application-example. Note: you will be in Mac Libraries and will need to switch to RPC-example folder. Now you can use [File / Quit] to leave THINK C.

4. **Run the program.**

Follow these steps to run the program you just created.

a. Double click on the Application-example icon you just created.

b. Type a symbol into the dialog box that appears and press Return. The symbol's property list will be displayed in a window.

c. Drag down the File menu and mouse Plist to examine another symbol or mouse Quit to quit the program.

Routines in RPC.lib

Routines for Initialization

InitializeRPC Routine

```
void InitializeRPC()
```

Initializes the RPC library. You must call `InitializeRPC` before calling anything else in the RPC library.

initialize_*remote_module_name*_server Routine

```
OSErr initialize_remote_module_name_server()
```

Every linked remote module server defines a C routine named `initialize_`*remote_module_name*`_server`, where *remote_module_name* is the name of the remote module, converted from Lisp syntax to C syntax (lower case letters, hyphens replaced by underscores). You must call each of these routines after calling `InitializeRPC`, to allow incoming RPC calls from the Ivory to find these servers.

initialize_predefined_remote_entries_server Routine

This server is in the RPC library, so it is always present. You must call it after calling `InitializeRPC`. See the section "initialize_*remote_module_name*_server Routine".

emb_agent_open Routine

```
OSErr emb_agent_open()
```

Establishes a bidirectional RPC channel between the host Macintosh and the embedded guest Ivory. `emb_agent_open` returns a standard Macintosh error code. If the value is not `noErr`, you should terminate the application.

Note: It is important to check this error code. Use the `ReportRPCOpenFailure` routine to report problems.

Routines for Termination

CloseRPC Routine

```
void CloseRPC()
```

Releases the bidirectional RPC channel between the host and the embedded guest Ivory.

MacIvory User Note: Before your application exits, it must call `CloseRPC` or the RPC channel allocated by `emb_agent_open` will remain permanently busy. Your `RestartApplications` routine must call `CloseRPC` and then call `emb_agent_open` again.

Routines for the Event Processing Loop

PollRPC Routine

```
void PollRPC()
```

Handles incoming requests and responses from Ivory. `PollRPC` must be called periodically, typically from your application's main event loop. `PollRPC` can call back to `BusyWait` and to RPC servers.

BusyWait Routine

```
void BusyWait(Boolean allow_rpc)
```

Your application must define the function `BusyWait`, to be called by the RPC library when it needs to wait. `BusyWait` should perform whatever polling you require once and then return; it should not loop.

The RPC library calls `BusyWait` repeatedly until the condition for which it is waiting is satisfied. `allow_rpc` is TRUE if the RPC system expects you to call it back, and FALSE if it does not. Call `PollRPC` if and only if `allow_rpc` is TRUE.

Symbolics UX User Note: Be careful that `BusyWait` is not run in a tight loop. See the section "Interfacing the Symbolics RPC Library to an Event Loop".

MacIvory User Note: `BusyWait` gives you an opportunity to implement whatever multiprocessing strategy you prefer. When using MultiFinder, `BusyWait` must call `GetNextEvent` or `WaitNextEvent` so that other Macintosh applications can run. In general, it is a good idea for `BusyWait` to support the mouse at least to the extent of allowing the Apple menu to be used, but it is not a good idea for `BusyWait` to run portions of your application that can make RPC calls to the Ivory.

MacIvory Example:

This example assumes:

1. The program has a global Boolean variable, WNEIsImplemented, whose value is set by the following code fragment:

```
#define             WNETrapNum      0x60
#define             UnImplTrapNum   0x9F

Boolean                 WNEIsImplemented;

WNEIsImplemented = (NGetTrapAddress (WNETrapNum, ToolTrap) != NGetTrapAddress
   (UnImplTrapNum, ToolTrap));
```

This variable tells the program whether to call `GetNextEvent` or `WaitNextEvent`.

2. The program has a function called ProcessEvent to process events.

Here's BusyWait:

```
void BusyWait (allow_rpc)
  int allow_rpc;
{
  EventRecord event;
  short mask;

        mask = (ColdLoadIsVisible ()) ?
                (everyEvent - keyUpMask - activMask)
                    : (everyEvent - keyDownMask - keyUpMask - autoKeyMask
                      - activMask - updateMask);

        if (!WNEIsImplemented)
                SystemTask ();

        if (allow_rpc)
                PollRPC ();

        MacIvoryTasks ();

        if ((WNEIsImplemented) ? WaitNextEvent (mask, &event, 2L, 0L)
                                                : GetNextEvent (mask, &event))
                ProcessEvent (&event);
```

UX Example:

```
/*** Include Files ***/

#include <sys/types.h>
#include <errno.h>
#include "RPC.h"
```

```
/*** Required by the RPC library ***/

void BusyWait(allow_rpc)
     int allow_rpc;
{
  int cc = 0, rpc_fd;
  fd_set read;

  /* Set up */
  rpc_fd = RPCAgentFileDescriptor();

  for (; cc == 0;) {

    /* Wait for something interesting to happen
     * We select for input ready on the RPC file descriptor.  In a more
     * complex program, there may be other interesting file descriptors
     * or a timeout here.
     *
     * We ignore EINTR, which means we were interrupted by a signal handler
     * being called, and EWOULDBLOCK, which SunOS has been seen to return
     * inappropriately in this situation.
     */
    FD_ZERO(&read);
    FD_SET(rpc_fd, &read);
    if ((cc = select(rpc_fd+1, &read, NULL, NULL, NULL)) < 0 &&
        errno != EINTR && errno != EWOULDBLOCK) {
      perror("BusyWait select");
      exit(-1);
    }

    /* If there is input pending on the RPC fd, and we're allowed
     * to make recursive calls to the RPC substrate, do so.
     */
    if (cc > 0 && FD_ISSET(rpc_fd, &read) && allow_rpc)
      PollRPC();
  }
}
```

Routines for RPC Error Handling in RPC.lib

RPCRemoteError Routine

```
RPCRemoteError (long *error-number)
```

Returns the remote error number of the last RPC call that failed. This routine is useful with the individual functions that access remote-error values to allow error handling.

ReportRPCOpenFailure Routine

```
Boolean       ReportRPCOpenFailure (OSErr error, Boolean embeddedP, char* host)
```

Reports a failure when opening an agent (that is, when calling emb_agent_open). The argument error is the return code from the call. For now, embeddedP should be TRUE and host should be 0L. Returns TRUE if unable to recover from the error; returns FALSE if able to recover from the error and if the agent is open.

ReportRPCCallFailure Routine

```
Boolean     ReportRPCCallFailure (Boolean fatalP, OSErr error,
              Boolean embeddedP, char *host)
```

Reports a failure from an RPC call. The argument error is the return code from the call. The argument fatalP should be TRUE or FALSE based on whether the error can be recovered from. For now, embeddedP should be TRUE and host should be 0L. If fatalP is FALSE, this routine returns TRUE if the user decides to give up and returns FALSE if he wants to try again. If fatalP is TRUE, the routine always returns TRUE.

Routines in MacIvory.lib

Note: Prototypes and type definitions are in MacIvory-Support.h.

Routines for Initialization

InitMacIvorySupport Routine

```
void InitMacIvorySupport(Boolean forMacIvory, int* YourResFile,
                           WindowPtr* ColdLoadWindow, MenuHandle*
                           IvoryControlMenu, Boolean* initedMacIvorySupport)
```

Initializes the MacIvory library and opens its associated resource file, *disk*:System Folder:Ivory:MacIvory-support.rsrc. The value of forMacIvory is TRUE, if you expect an Ivory coprocessor to be present in the system. It is FALSE if the library is present in your application, but you are not using the Ivory. In MacIvory programs, forMacIvory should always be TRUE.

The remaining arguments are addresses of variables that receive return values:

YourResFile is set to the reference number of the application resource file. ColdLoadWindow is set to point to the Macintosh window that contains the cold-load stream. IvoryControlMenu is set to a handle to the Ivory menu, which you should add to the menu bar. The latter two variables are not set if forMacIvory is FALSE. initedMacIvorySupport is set to TRUE if the library was initialized and will have to be shutdown by TermMacIvorySupport later.

InitMacIvory Routine

```
void InitMacIvory(Boolean NeedsLispRunning)
```

Checks on the status of the Ivory coprocessor, initializes the coprocessor if it is uninitialized, and boots it, if needed. NeedsLispRunning is TRUE if Lisp must be running. It is FALSE if it is okay for just the IFEP to be running.

Routines for Termination

TermMacIvorySupport Routine

```
void TermMacIvorySupport(Boolean forMacIvory)
```

Shuts down the MacIvory library. If the Ivory is not running, relinquishes control of the network interface so other Macintosh applications can use it. Call this whenever an Ivory-using application exits. Supply the same value for forMacIvory as in your call to InitMacIvorySupport.

ExitMacIvoryApplication Routine

```
void ExitMacIvoryApplication()
```

You must define this routine in your application. This routine is called by the library if the application must be terminated unexpectedly. It should call TermMacIvorySupport if the library is initialized. It should call CloseRPC if an agent was opened. In all cases it should also cleanup after the appplciation and ExittoShell.

OKtoStopMacIvory Routine

```
Boolean OKtoStopMacIvory()
```

Returns TRUE if it is okay to quit the application now; either the Ivory is not running or the user has confirmed that it is okay via an alert box. This routine is usually called in your handler for the "Quit" menu item.

RestartMacIvoryApplication Routine

```
void RestartMacIvoryApplication()
```

You must define a routine named RestartMacIvoryApplication, which is called with no arguments from MacIvoryTasks when the Ivory system is booted or restarted. RestartMacIvoryApplication should reset whatever is appropriate to reset in your application. You must call CloseRPC in this function and then open a new channel.

RestartMacIvoryApplication can either return to its caller or perform a longjmp to the start of your application.

Routines for the Event Processing Loop

MacIvoryTasks Routine

```
void MacIvoryTasks()
```

MacIvoryTasks must be called periodically, typically from your application's main event loop. It handles reset requests from Ivory and maintains the cold-load-stream window. MacIvoryTasks can call back to ExitMacivoryApplication or NoteMacIvoryStateChange.

MacIvoryEvent Routine

```
Boolean MacIvoryEvent(EventRecord* Event, long* MenuAndItem)
```

Your application's event processing loop should call MacIvoryEvent on every event, before processing it.

MacIvoryEvent takes care of the Ivory menu and the cold-load-stream window. If MacIvoryEvent returns TRUE, it has fully handled the event and the application can ignore it. Otherwise, the application should handle the event normally. However, if the event is mouse-down in the menu bar, MacIvoryEvent has already called MenuSelect and set the variable addressed by MenuAndItem to the result, so the application should not call MenuSelect again.

NoteMacIvoryStateChange Routine

```
void NoteMacIvoryStateChange (enum MacIvoryStateTransition state)
```

NoteMacIvoryStateChange is called by the library whenever Ivory's state changes, for instance, when it shuts down or starts Lisp. You must define this routine in your application. Posssible state transitions are:

Value of State	*Meaning*

MacIvoryHasBroken	Library has detected a fatal situation. Ivory may be unusable.
FEPHasStopped	IFEP has shutdown. Ivory is no longer running.
FEPIsRunning	IFEP is booted.
LispIsRunning	Lisp has been cold or warm booted.
LispHasStopped	Lisp has halted or otherwise encountered a fatal condition.

Control Routines in MacIvory.lib

ColdLoadVisible Routine

```
Boolean    ColdLoadIsVisible ();
```

Reports whether the cold load window is displayed. Returns a value of TRUE if the cold load window is currently on display (the frontmost window). Returns a value of FALSE if the cold load window is currently hidden.

IsColdLoadWindow Routine

```
Boolean             IsColdLoadWindow (WindowPtr Candidate);
```

True if `Candidate` is the cold load window.

IsNetworkEnabled Routine

```
Boolean             IsNetworkEnabled ();
```

True if Ivory has attached the Ethernet interface.

MacIvoryIsRunning Routine

```
 Boolean            MacIvoryIsRunning ();
```

True if the Ivory coprocessor is running either Genera or IFEP.

MacIvoryIsRunningLisp Routine

```
Boolean             MacIvoryIsRunningLisp ();
```

True if the Ivory coprocessor is running Genera.

RunningInBackground Variable

```
Boolean  RunningInBackground
```

A value of true indicates the application is running in the background under MultiFinder.

A value of false means the application is the selected application under MultiFinder or is the only application under the Finder.

myProgramID Variable

```
long  myProgramID
```

A unique ID assigned by the MacIvory support library to the application.

Routines for the Ivory Menu in MacIvory.lib

Calling one of these entry points does exactly what the equivalent Ivory menu item does. If the menu item puts up an alert box asking for confirmation to complete an action, the equivalent entrypoint does the same. If the menu item puts up an alert box when and operation does not finish in time or fails, so does the equivalent entrypoint.

ColdBootFEP Routine

```
ColdBootFEP ();
```

Cold boots the IFEP. This is equivalent to the "Cold Boot FEP" item in the Ivory menu.

ColdBootLisp Routine

```
ColdBootLisp ();
```

Cold boots Lisp. This is equivalent to the "Cold Boot Lisp" item in the Ivory menu.

ContinueLisp Routine

```
ContinueLisp ();
```

Switches control from the IFEP to Lisp. This is equivalent to the "Transfer to Lisp" item in the Ivory menu.

DisableNetwork Routine

```
void        DisableNetwork ();
```

Disables the Ivory's use of the Ethernet interface.

EnableNetwork Routine

```
void        EnableNetwork ();
```

Re-enables the Ivory's use of the Ethernet interface.

HideColdLoad Routine

```
void        HideColdLoad ();
```

Hides the cold load window.

RestartFEP Routine

```
RestartFEP ();
```

Warm boots the IFEP. This is equivalent to the "Restart FEP" item in the Ivory menu.

RestartLisp Routine

```
RestartLisp ();
```

Warm boots Lisp. This is equivalent to the "Restart Lisp" item in the Ivory menu.

ShowColdLoad Routine

```
void        ShowColdLoad ();
```

Call this to show the cold load window on the display.

ShutDownIvory Routine

```
ShutDownIvory ();
```

Halts the Ivory. This is equivalent to the "Shut Down" item in the Ivory menu.

StopLisp Routine

```
StopLisp ();
```

Switches control from Lisp to the IFEP. This is equivalent to the "Transfer to FEP" item in the Ivory menu.

Developing User Interfaces with MacIvory

How the MacIvory User Interface Works

The MacIvory system user interface substrate is designed with flexibility in mind, so that applications can make maximally effective use of both of the co-processor systems. As much as possible, we have tried not to constrain decisions about user interface design by imposing fundamental limitations in the system. Choices can be made on an application-by-application basis by the application programmer. In some cases, it may even be possible to defer decisions to the final end-users by offering a variety of styles from which they choose.

There are two main guiding forces in the way that MacIvory works. These are:

1. Compatibility with other Symbolics products, especially the previous stand-alone proprietary platforms.

2. Good, smooth integration with the Macintosh operating system and its user interface guidelines and conventions.

These forces often work at cross-purposes. It sometimes is not possible to fulfill both sets of constraints simultaneously within a given interface. For this reason, we allow the the application programmer to decide how to balance these constraints, and have made every effort to make this process as easy as possible.

For instance, the generic file system model is used to provide access to both the native Macintosh file system and a local LMFS file partition on the Ivory partition of the disk. It is up to the user to decide where to store his or her files.

Most important, perhaps, is how these principles apply to the user interface.

The user interface is controlled by an application which runs on the Macintosh on behalf of the Ivory. The Ivory communicates to the Macintosh using a standard Remote Procedure Call facility (RPC) through a shared memory channel. The workhorse of this communication is a remote console protocol, by which the Ivory requests that the Macintosh draw lines, rectangles, characters, and the like. These requests are handled asynchronously; that is, the Ivory does not wait for the Macintosh to finish drawing. Higher-levels requests, such as reading user responses via a dialog window, obviously must wait.

All drawing is handled by the Macintosh. Among other things, this means that any display monitor which works with QuickDraw will work with MacIvory. It also means that there is no array in Lisp virtual memory, which is mapped to an exposed window's portion of the display's frame buffer.

The compatibility of the MacIvory system is not limited by the exact capabilities that the Macintosh provides in its high-level toolbox.

- The Apple user interface guidelines recommend very strongly against warping the mouse (cursor). However, a number of existing Genera applications rely on this capability for correct functioning. Therefore, the MacIvory provides the capability and it is up to the application programmer to only use it wisely.

- When copying from a small image to a larger one, the QuickDraw CopyBits procedure will scale the image up linearly. The Genera **bitblt** primitives are defined to replicate the source image in these cases. MacIvory provides a fully compatible interface for copying from a raster array to the screen, with replication. If necessary, more than one CopyBits call is performed to produce the desired result.

- The QuickDraw line drawing interface does not provide a way of suppressing drawing of the final endpoint. The polygon drawing interface is not completely compatible with Genera's draw-triangle, which was specifically designed to allow triangles with common edges to abut seamlessly. Therefore interfaces to Genera-compatible line and triangle drawing are provided which work in terms of lower-level QuickDraw primitives, rather than LineTo or PaintPoly.

- QuickDraw has a relatively fast character drawing entry. However, the normal Macintosh screen fonts are not compatible with Genera fonts. Therefore, MacIvory provides copies of all the standard Genera fonts in a format which the Macintosh can use for fast drawing.

- QuickDraw patterns can only be 8x8 bits, no more, no less. The Genera graphics substrate provides more general pattern and stipple patterns. Therefore, the QuickDraw patterns are not normally used for stippling when strict compatibility is enabled.

Of course, all this compatibility comes with a performance penalty. Many applications are prepared to trade strict compatibility for improved performance. For this reason, we provide access to drawing capabilities which more nearly match those provided natively by the host, while still maintaining the Genera user interface style.

- If an application is using the Genera **graphics:draw-xxx** primitives, a single form is provided which enables use of QuickDraw's high level entries to accomplish drawing. This device independence was one of the major criteria in the design of Genera's unique graphics substrate.

- Applications using the lower-level **:draw-line** and **:draw-triangle** messages can still enable a mode where these use QuickDraw lines and polygons.

- A Genera-style interface application can be configured to use the Macintosh's native fonts instead of the default Genera fonts copied to the Mac, or to use smaller fonts to account for the smaller screens with a lower pixel density that are common on Macintoshes. If the application is using character styles, this change is mostly invisible. This independence was one of the major criteria in the design of Genera's unique characters substrate.

- Bit arrays which need to be copied to and from the screen can be stored on the Macintosh side and drawn into remotely there. The Macintosh application which runs on behalf of Ivory will also swap these back and forth from the disk as need be, so there is no practical limitation on their size or number, even though the Macintosh does not have virtual memory.

Moving further away from compatibility, it is possible for a Genera application to make use of the Macintosh user interface toolbox, which comes with a well established reputation for ease-of-use.

This access can be made directly by using the Lisp interface to the toolbox functions. However, we imagine that most Genera programmers are unaccustomed to having to deal at this low a level, and would prefer to use higher level-interfaces.

Compatibility is also provided in user interface peripherals.

- A number of large displays for the Macintosh approximate the size and pixel density of the standalone Genera workstation screens.

- The Apple extended keyboard can be used to input all the characters that can be typed on the Symbolics keyboard. Of course, some of these combinations are awkward. For this reason, there is an option for a Symbolics keyboard which interfaces to the Apple Desktop Bus and is therefore fully usable with the Macintosh as well. Alternatively, a user can customize the mapping for the Apple extended keyboard into the Symbolics keystrokes using an interactive program. For instance, we have found a wide difference in opinions as to how the three meta shift keys should be mapped into the three primary Symbolics shifts.

- A three-button mouse option which connects to the ADB is provided for those applications which make heavy use of the middle and right buttons, which require keyboard assistance without it.

Basic Color Support in MacIvory

If the host has color display hardware, you can draw in color by using the **:color** or **:gray-level** arguments to the **graphics:draw-***xxx* functions. The value of :color is a symbol that names a color (one of **:black**, **:red**, **:green**, **:blue**, **:cyan**, **:yellow**, **:magenta**, **:white**), a list (*red green blue*) where each element is a number between 0 and 1, inclusive, or a color object created by **color:make-color**. The value of **:gray-level** is a number between 0 and 1, inclusive.

See the section "Pattern Options".
See the option **:color**.
See the option **:gray-level**.

In addition, when the host has color display hardware, the Genera window system can run in color mode or in monochrome mode. You must have the Ivory-Color-Support system loaded to use color mode. Both modes can draw in color. The difference involves saving and restoring displayed images when switching windows. In monochrome mode, only one bit per pixel is saved, so when you switch windows

any colored drawings lose their color. In color mode, the complete image is saved including the color information. Color mode comes at a cost: a much larger amount of information must be transmitted to and from the screen. For example, in 8-bits-per-pixel mode switching windows requires eight times as much information transmission, and off-screen bit arrays require eight times as much host memory. Depending on the hardware configuration, storing and transmitting the extra information can slow down response to an unacceptable level. For this reason, the default mode is monochrome mode even when color display hardware is present. See the section "Configure MacIvory Application Command" for information on how to enable color mode.

The above considerations also apply when the host has display hardware that can display shades of gray; that is, more than just black and white, but less than full color. In monochrome mode, the gray level is converted to black or white when switching windows; in color mode, the gray level is preserved at a cost in performance determined by the number of bits per pixel.

color:make-color *(*&key *:red :blue :green :intensity :hue :saturation* &allow-other-keys*)* *Function*

Creates a color object. Color objects can be used with the **:color** argument to the **graphics:draw-***xxx* family of functions.

The arguments are numbers between 0 and 1 inclusive, defaulting to 0. To specify a color in the RGB color model, specify one or more of **:red**, **:green**, and **:blue**. To specify a color in the IHS color model, specify one or more of **:intensity**, **:hue**, and **:saturation**.

Higher-level Interfaces to the Macintosh Toolbox

If you are not concerned with compatibility among platforms, you may wish to convert your program to have a Macintosh-style user interface. A specialized version of **dw:define-program-framework** is provided with capabilities that implement the Macintosh-style user interface. These capabilities include:

- Use of hierarchical pull-down menus from the menu bar to implement program command levels of menus and subcommands.

- Stream output to Macintosh windows, either in realtime, or via a picture record structure which gives a window that can be scrolled entirely on the Macintosh side.

- Use of dialog boxes for **dw:accepting-values**.

The major pieces of this facility are:

- **dw:define-remote-program-framework**, which controls layout of the menu bar among command menu items.

- **dw:with-remote-accepting-values**, which gives access to dialog boxes for accepting-values. This is also invoked automatically if you give **:menu-accelerator** for a command.

- **dw:with-output-to-viewer**, which allows stream output to a Macintosh window.

If you plan to use these facilities, you may wish to consult your MacIvory customer support contact. Limited resources and the requirement of compatibility among all system products did not permit the actual conversion of any Genera applications to have Macintosh-style user interfaces.

For examples of the use of these facilities, see the section "Example of Converting an Application for MacIvory".

Functions for Creating Macintosh-style Interfaces

dw:define-remote-program-framework *name* &body *options* &key *(:command-definer* **t***) :menu-level-order :top-level :command-table :inherit-from :state-variables*
Special Form

A subset of **dw:define-program-framework**, which implements a program that can be run using the Macintosh-style user interface. Each menu level specifies a column in the pull-down menu. Releasing the mouse on a given item is equivalent to clicking on that item in the Genera style interface. Normally, menu items are filled in by means of the **:menu-accelerator** option to a program command definition.

The **:name**, **:command-definer**, **:top-level**, **:command-table** , **:inherit-from**, **:state-variables**, and **:pretty-name** options are just as for **dw:define-program-framework**.

The **:selectable** option is slightly different, in that it can also be a list of remote system types. For instance, **:selectable** (**:mac**) would mean that the program could be started up as a MacIvory application, but would not appear in the Select Activity menu. **:selectable t** means that a remote host can start up the application using its interface style, and Select Activity will start up the same application using the Genera style (even on a MacIvory, when it is running the Genera application).

The **:menu-level-order** option represents a list of lists (evaluated) of item names (or **nil** for empty slots) specifying how the items in specific columns are laid out. If you do not give this explicitly, the order of columns is unpredictable, and the order within columns tends to be alphabetical.

Example:

```
:menu-level-order '(,macintosh-internals::*standard-remote-viewer-file-menu*
                    ,macintosh-internals::*standard-edit-menu*
                    "Lookup")
```

dw:remote-program *Flavor*

The basic flavor included in programs running with the host user interface style.

dw:remote-program-quit &optional *(program* **dw:*program***)* *Function*

Exits the host application corresponding to this program. On the Macintosh, this closes all its windows back to the icon.

Normally, this is accessed with the Quit command in the remote-quit-commands command table (which you can inherit). But sometimes a program wishes to do it for other reasons.

dw:remote-program-p *local-program* *Function*

Returns **t** if the program is using the host user interface. Useful for large-scale conditionalization of behavior in a program which also works with the Genera-style interface.

dw:with-remote-accepting-values &optional *(stream* ***query-io****)* &key *(:program* **dw:*program***) *:prompt* *Function*

Like **dw:accepting-values**, but uses the host dialog style if the program uses the host interface style. On the Macintosh, the host dialog is constructed by classifying each query in the body (that is, each call to **accept**) as one of the following:

- A boolean choice
- An enumeration from an explicit set
- An arbitrary string

These are laid out (automatically) in a dialog menu as check boxes, radio buttons, and text fields, respectively.

Most often, this capability is accessed for the command arguments to a program command given with **:menu-accelerator**, when the user releases the mouse on the corresponding item.

dw:define-remote-program-command *Function*

Like **dw:define-program-command**. Almost always accessed via the **define-*xxx*-command** macro generated by the **:command-definer** option to **dw:define-remote-program-framework**.

If **:menu-accelerator** is given, a menu item is added to the host command menus for this command. If the command takes arguments, selecting this item gives a dialog box for those items. Otherwise, it runs the command immediately.

The format of **:menu-level** is extended slightly. The elements of the list are names of levels in the Genera style interface (often keywords), or lists of a remote system type and level within the interface for that system. The next example puts the

item in the main menu, if Genera-style, and in the Commands pull-down menu, if Macintosh style.

```
(define-my-command (com-show-doc :menu-accelerator "Show"
                                  :menu-level (:top-level (:mac :commands))
                                  :keyboard-accelerator #\s-S)
  ...)
```

dw:with-output-to-viewer (&optional *stream* &rest *args*) &body *body* *Special Form*

Adds output from *body* to the program's display. If the program is using the Genera-style interface, the primary pane has its contents replaced with those obtained by the body. The window's label is set to the title, if any. The other options are ignored.

If the program is running using the Macintosh-style interface, a new window is created. The options work as follows:

:progress-note If given, the watch mouse icon is shown while the output is being computed.

:picture-p If **nil**, the window does not have any contents initially and nothing is regenerated automatically when the screen area is disturbed on the Macintosh side.

If non-**nil**, the output is buffered into picture records that are remembered on the Macintosh side. The window can then be scrolled and refreshed entirely by the host.

:title Gives the string for the title bar.

:width, **:height**, **:left**, **:top**, **:right**, **:bottom**
These specify the location and size. The default is positioned slightly offset from other windows and as large as picture contents, if any, or the entire screen if not.

:color-p Makes a color Macintosh window rather than an old-style one. This allows use of the **:color** keyword within the output body to work properly with **graphics:draw**-*xxx*.

:buffer-screens Exposes the window as soon as the first screenful of output has been collected. If the window is then scrolled remotely past the end of the output known to the Macintosh, the next screenful is requested of the guest.

:buffer-ahead-screens
Used with **:picture-p t**, this option specifies the number of screenfuls to keep buffered past the visible end. This makes scrolling one screen at a time not pause as often, assuming the user stops to read each screenful.

If **present** or **dw:with-output-as-presentation** is used within the output body, the corresponding output can be sensitive when using the host user interface. For the Macintosh, a rectangle is inserted containing sensitive items while the mouse is held down, and the corresponding action taken when the mouse is released. These actions are defined in the normal way with **define-presentation-translator**. At present, only translation to program commands works (that is, there is only one valid input context).

Program: Macintosh-internals:remote-quit-commands

Defines the Quit command, with the s-Q command accelerator. You can inherit this program and its command table to get the command.

Program: Macintosh-internals:remote-viewer-commands

Defines commands useful for dealing with stacks of windows generated by **dw:with-output-to-viewer**:

Close (s-W) closes the front window.

Close All closes all windows for this application.

This program also depends on the remote-quit-commands program, which means you get the Quit command also when you include it.

For example,

```
(dw:define-remote-program-framework my-program
  :selectable (:mac)
  :menu-level-order `(,macintosh-internals::*standard-remote-viewer-file-menu*
                      ,macintosh-internals::*standard-edit-menu*
                      "My commands")
  :inherit-from (macintosh-internals::remote-viewer-commands)
  :command-table (:kbd-accelerator-p t :inherit-from '("remote-viewer-commands"))
  )
```

Macintosh Notes for dw:define-subcommand-menu-handler

If you use the function **dw:define-subcommand-menu-handler** in a Macintosh-style program, you get a hierarchical pull-down menu. For example,

```
(dw:define-subcommand-menu-handler "More Tests" test ((:mac :test))
        (:mac :more-tests))
```

Creating a Macintosh Application That Runs an Ivory-based Program

This section describes how to create a Macintosh application that provides the user interface for your Ivory program. Before doing so, you must have created the Ivory portion of your application with **dw:define-remote-program-framework**. You must also load your program framework into Genera.

The Configure MacIvory Application command creates a Macintosh program using the values you provide. This program provides the user interface. See the section "Configure MacIvory Application Command". A double click on the resulting Macintosh application icon will communicate with the Ivory through RPC and run your application.

Creating a Macintosh Application That Uses the Genera Window System

To create a Macintosh application that runs an Ivory-based program under the Genera window system, rather than the Macintosh's own window system, use the following procedure. Configure a copy of the Symbolics-supplied "Genera" Macintosh application that uses the "Start Screen" initial application command to start up your application. After filling out the first four lines of the form, click on "*a GENERA command*", type Start Screen followed by a space, and type in the arguments to the command. If you press `m-COMPLETE` after Start Screen, you will see a menu of the command arguments. For most purposes you can press `RETURN` After the first two arguments.

This works for any application that can be started by the Select Activity command. For example, to create a Macintosh application that runs the Zmacs editor:

```
:Configure Macivory Application
From file: HOST:DSK:Genera
To file: HOST:DSK:Editor
Application: None  Genera
Version: 1.0d0
Agent: Tcp  Serial  Reliable-Serial  Emb
Initial application command: None  Start Screen Genera fonts installed
on Mac "Zmacs"
```

Creating and Using Macintosh Dialogs

Sometimes it may be necessary to exert more control over the appearance of Macintosh dialogs than **dw:with-remote-accepting-values** allows. By dealing with things at a somewhat lower level, you can do it.

You create a *dialog maker*, which is an instance of flavor **mtb:dialog-item-maker**, and use various generic functions with it. You may add dialog items, specifying their individual type, location, and so on, along with a symbolic query id. You may specify the edges and window type of the dialog window itself. And, finally, your dialog maker can interact with the user with a modal dialog; the user's choices, selections and entries are returned in plist form with the query ids.

Make a *dialog maker* with **mtb:make-dialog-item-maker**. Add dialog items to it procedurally with **mtb:add-dialog-button**, **mtb:add-dialog-text**, **mtb:add-dialog-edit**, **mtb:add-dialog-check**, **mtb:add-dialog-radio**, **mtb:add-dialog-pict**, and **mtb:add-dialog-line**. If you have the appropriate data structure in hand, add dialog items declaratively with **mtb:add-several-dialog-items**. Control dialog item clustering with **mtb:in-dialog-cluster**. Specify aspects of the dialog window itself (as opposed its the dialog items) with **mtb:set-dialog-face**.

Having specified the dialog items and window, use **mtb:do-modal-dialog** to show the window and handle the modal dialog interaction.

Dialog Item Clusters

Dialog items within a dialog can be grouped by *cluster*. Presently, clusters are useful only for mutually exclusive radio buttons: in such a cluster, when you click on one radio button to turn it on, the others in the same cluster are turned off.

The Lisp model for every dialog item contains a cluster id. Dialog items are considered to be in the same cluster if they have the same cluster id. Cluster ids are compared with **eql**.

mtb:add-dialog-button *dialog-item-maker rect title* &key *:cluster :active :check :query-id* *Function*

Adds a Button dialog item to *dialog-item-maker*.

See the section "**add-dialog-item** Arguments".

mtb:add-dialog-check *dialog-item-maker rect title* &key *:active :report :cluster :oversee-cluster :state :query-id* *Function*

Adds a checkbox dialog item to *dialog-item-maker*.

See the section "**add-dialog-item** Arguments".

mtb:add-dialog-edit *dialog-item-maker rect* &key *:text :query-id :required :oversee-cluster* *Function*

Adds an editable text item to *dialog-item-maker*. Initial contents may be specified with **:text**.

See the section "**add-dialog-item** Arguments".

mtb:add-dialog-line *dialog-item-maker rect* *Function*

Draws a line on the dialog window, perhaps to separate groups of dialog items. The line is drawn with a 50%-density stipple, with a pen two pixels wide, from the top left of *rect* to its bottom right.

mtb:add-dialog-pict *dialog-item-maker rect resource-id* &key *:active :check :cluster :query-id* *Function*

Adds a PICT dialog item to *dialog-item-maker*. *resource-id* is the PICT resource id of the picture. Your application will have to arrange for the appropriate resource file to be open when the PICT is to be drawn.

See the section "**add-dialog-item** Arguments".

mtb:add-dialog-radio *dialog-item-maker rect title* &key *:active :report :cluster :oversee-cluster :state :query-id* *Function*

Adds a radio button dialog item to *dialog-item-maker*.

See the section "**add-dialog-item** Arguments".

Radio buttons in the same cluster are mutually exclusive: when you click on one to turn it on, the others in the same cluster are turned off. See the section "Dialog Item Clusters".

mtb:add-dialog-text *dialog-item-maker rect text* *Function*

Adds static text to *dialog-item-maker*.

See the section "**add-dialog-item** Arguments".

mtb:add-several-dialog-items *dialog-item-maker spec-list* *Function*

Adds several dialog items to *dialog-item-maker* under control of *spec-list*.

spec-list is essentially an abbreviation for separate calls to **add-dialog-***xxx*. The first element of each spec in *spec-list* is a keyword, as in the table below. The rest of each element becomes the rest of the args to the corresponding function.

Keyword	*Function*
`:button`	**mtb:add-dialog-button**
`:check`	**mtb:add-dialog-check**
`:edit`	**mtb:add-dialog-edit**
`:line`	**mtb:add-dialog-line**
`:pict`	**mtb:add-dialog-pict**
`:radio`	**mtb:add-dialog-radio**
`:text`	**mtb:add-dialog-text**

In addition, you can use the keyword **:cluster** as in (`:cluster` *cluster-name* `&rest` *spec-list*) to abbreviate calls to **mtb:in-dialog-cluster**.

For example, see the file SYS:EMBEDDING;MACIVORY;TOOLBOX;EXAMPLES;XXX.

mtb:do-modal-dialog *program dialog-items* *Function*

In the context of the remote program *program*, does the modal dialog specified by the dialog item maker *dialog-items*. Exposes the dialog window, and repeatedly calls ModalDialog with a Lisp event filter callback. Handles clicks on check boxes and clustered radio buttons. Handles interaction with TextEdit. Returns when the user clicks on either of the first two dialog items (assumed to be buttons named "OK" and "Cancel"), or presses `RETURN`, `ENTER` or `END`.

Also terminates when the user preses `COMPLETE` (the key labelled "HOME") or `HELP` in an editable text field.

The value returned by **mtb:do-modal-dialog** is a plist of alternating query-ids and values. Each query-id comes from the **:query-id** argument to the **add-dialog-***xxx* which created the dialog item. The corresponding values are strings (for editable text items), or **t** (for check boxes and radio buttons, when checked). Although a cluster of mutually exclusive radio buttons would be better modelled as a choice from an enumeration, there is no automatic support for returning this as a value for a specific query.

mtb:in-dialog-cluster *(dialog-item-maker cluster-id)* &body *body* *Macro*

Within *body*, provides implicit **:cluster** *cluster-id* arguments for calls to **mtb:add-dialog-button**, and so on. See the section "Dialog Item Clusters".

mtb:make-dialog-item-maker *Function*

Makes a dialog item maker.

See the section "Creating and Using Macintosh Dialogs".

mtb:set-dialog-face *dialog-item-maker* &key *:bounds :title :proc-id :go-away* *Function*

Specifies the appearance for a dialog window (as opposed to that of its dialog items).

dialog-item-maker

keywords	**:bounds**, **:title**, **:proc-id**, and **:go-away**.
:bounds	A list (left top right bottom) or a Rect octet structure. Specifies the edges of the window, defaulting to (40 40 440 340).
:**title**	A string, Appears in the window's title bar, if any.
:**proc-id**	A small integer. Controls the shape of the dialog window. See *Inside Macintosh*, page I-273. For example, to get a plain rectangular box, use `(mtb:cconstant plainDBox)`; for a "rounded-corner" window with black title bar you would use `(mtb:cconstant rDocProc)`.

:go-away A boolean. Specifies whether there should be a go-away box in the top left of the corner of the window.

add-dialog-item Arguments

mtb:add-dialog-button, **mtb:add-dialog-text**, **mtb:add-dialog-edit**, **mtb:add-dialog-check**, **mtb:add-dialog-radio**, **mtb:add-dialog-pict**, and **mtb:add-dialog-line** all take the current *dialog-item-maker* as first argument. Several of them share other arguments:

rect Specifies the edges of the dialog item, as a list (left top right bottom) or a Rect octet structure.

title A string. Specifies the text, if any, displayed with the dialog item.

query-id May be any Lisp object. Identifies the query in the plist returned by **mtb:do-modal-dialog**.

cluster Any Lisp object. Groups dialog items together. See the section "Dialog Item Clusters".

Low-level Interfaces to the Macintosh Toolbox

Overview of the Low-level Interfaces

Lisp programs on the MacIvory have complete access to the Macintosh User Interface Toolbox by means of the Remote Procedure Call (RPC) mechanism. MacIvory provides predefined remote entries that you can use to access Macintosh toolbox routines. There is no need to write your own remote entries. These routines can be found in the **mtb** (**mac-toolbox**) package.

This means, for example, that if you are running an application on MacIvory and you want to make use of the Macintosh StandardFile package to prompt the user for the name of a file in the Macintosh file system, you can call the Lisp function **mtb:_sfgetfile** (the underscore is a naming convention that distinguishes the Lisp versions of Macintosh toolbox routines).

The code on the Macintosh side that enables this is stored in a resource file recognized by the RPC mechanism. When a program using RPC on the Macintosh encounters a call to one of the toolbox routines it will load the necessary resource and run the routine.

To allow the greatest possible functionality, we have implemented all the routines described in volumes I through V of *Inside Macintosh* with the exception of Appletalk Manager routines. For a complete list, see the section "Lisp Functions That Access the Macintosh Toolbox".

Note that this list includes routines that may make no sense to call while running the Genera application on the Macintosh. For example, calling the Lisp function **mtb:_exittoshell** would cause the Genera application to exit, probably without doing necessary cleanup.

Predefined Types

Mac type	Lisp type
struct *name*	octet-structure — available by saying **(mtb:make-name ...)** as in (**make-point :x 10 :y 20**) or (**make-sfreply**). These provide accessors and init options for each of the slots.
array	A Lisp vector
pointer	fixnum (the Macintosh address)
integer	fixnum
floating points	float
boolean	**nil** or non-**nil**

Conventions Used by the MacIvory Toolbox Interface

Routine Names

Lisp function names are identical to the Pascal names mentioned in *Inside Macintosh*. The only differences are that Lisp functions are identified by a preceding underscore and are not case sensitive.

For example, the Macintosh function GetResource is called with the Lisp function **mtb:_getresource.**

Arguments and Values

Another difference between the Lisp versions of these routines and the *Inside Macintosh* Pascal versions is the handling of VAR parameters. The rule of thumb is that any VAR parameters that are structures (vectors in Lisp) of a size greater than one longword are overwritten with the value returned from the remote call. If there are other values to return, as in the case when a toolbox routine is a function, they are returned as multiple values with the function value first and the VAR parameters following in order of passing.

When in doubt, remember that the command Show Function Arguments (c-sh-A) will show you the arguments and the values returned for the toolbox functions. The argument names are the same as those used in *Inside Macintosh* except in cases where "in" and "out" have been appended to the argument name to indicate that the argument will be overwritten.

Error Signaling

Those Toolbox routines that are liable to generate an error, that is, those that return a value of type OSErr, use the RPCError facility to signal errors in the Lisp function that calls them. This means that if you are expecting an error result from a function, and would like to take some action on encountering an error, you can use the Lisp condition handling mechanism to react to the error. The exceptions to this are those functions such as **_memerror** and **_reserror** whose sole job is to check for error conditions and functions such as **_sysenvirons** that can return error code and real data.

MemError, ResError, and PrError

To avoid having to do two RPC calls for every memory manager, resource manager, or printing manager routine — one for the routine and one to check for errors — the interface to these managers makes the appropriate call for you and signals an error when needed.

You still need to do error handling in your RPC calls, but the appropriate error function is checked for each call to the memory manager, resource manager and printing manager routines.

Predefined Macintosh Types

In order to make writing RPC remote entries and using the toolbox interface easier, MacIvory software defines a number of the Macintosh types using the RPC data type extension mechanism (see the section "The RPC Data Type Extension Language"). These types include most of the types described in *Inside Macintosh*, Volumes I through V. Understanding the correspondence between the types on the Macintosh and the associated Lisp types is important when using the toolbox interface.

There are two categories of types: simple types and compound types. Simple types mostly correspond directly to Macintosh types in name and fun ctionality. Compound types are represented in Lisp as structures that correspond to Macintosh Record types. When a Toolbox routine is expecting a simple type like an integer, pointer, or floating-point, you can use the corresponding Lisp type. When the argument is a Macintosh Record, you should create the structure in Lisp, as described below, and use that as the argument.

Simple Types

For most types the correspondence between the Lisp and the Macintosh types is simple. For example, if a Toolbox routine requires a specific numeric type such as Fixed for one of its arguments, you can just pass the routine a Lisp floating-point number, as in this example:

```
(mtb:_spaceextra 1.3)
```

RPC translates the Lisp floating-point into the appropriate representation for the Macintosh type, in this case the Fixed type.

Routines expecting integer types such as long and word can be called by just passing in a Lisp integer of the appropriate size, as in this example:

```
(mtb:_textsize 12)
```

Macintosh pointer and handle types are 32-bit integers and can be manipulated as Lisp integers. For these types RPC takes care of all the data transport issues: converting from Lisp types to Macintosh types, doing the byte-swapping when needed, and converting the answer back from the Macintosh representation associated Lisp type.

Compound Types

There are a significant number of compound types defined in *Inside Macintosh*, Volumes I through V, that the Macintosh Toolbox routines need to be able to accept as arguments. In order to allow you to manipulate objects of types, and to use them as arguments to Lisp Toolbox routines, there is a corresponding set of Lisp structure types. These types are implemented as octet-structures, and provide associated constructor and accessor macros.

Using Octet Structures

Octet-structures are **art-8B** (unsigned-byte 8) arrays with their own set of functions for accessing and manipulating slots. Using octet-structures allows you to create and manipulate a structure in Lisp that can then be sent across RPC with virtually no further translation. That is, it provides a uniform representation of the data across both machines. This is a convenient way to manipulate data in Lisp in the same byte format that is used on the Macintosh.

Here's a simple example of using one of these octet-structures, **mtb:datetimerec**, to set and read the time on the Macintosh. Notice that the constructor and accessor naming convention is the same as for Lisp structures.

The Macintosh record DateTimeRec has slots for year, month, day, hour, minute, second, and day of week. This data structure might be defined in C as follows:

```
typedef         struct
        {
        int     year;
        int     month;
        int     day;
        int     hour;
        int     minute;
        int     second;
        int     dayOfWeek;
        } DateTimeRec ;
```

Because there is a corresponding octet-structure definition in Lisp, you can set the time on the Macintosh with:

```
(let ((date (macintosh-internals:make-datetimerec
             :year 1988
             :month 10
             :day 28
             :hour 13
             :minute 15
             :second 0
             :dayofweek 6)))
  (mtb:_setdatetime (mtb:_date2secs date)))
```

Notice how you can construct a DateTimeRec in Lisp by using the constructor defined by the octet-structure. You can use accessors like **mtb:datetimerec-hour** to look at or alter the slots of the structure.

Another common use of these structures is to get values back from calls to the Macintosh toolbox. For example, to read the date:

```
(let ((date (macintosh-internals:make-datetimerec)))
  (mtb:_secs2date (mtb:_getdatetime) date)
  (rpc:describe-octet-structure 'macintosh-internals:datetimerec date 0))

YEAR[0]:                                1990
MONTH[2]:                               10
DAY[4]:                                 28
HOUR[6]:                                13
MINUTE[8]:                              25
SECOND[10]:                             51
DAYOFWEEK[12]:                          6
NIL
```

In this case you first create the structure to hold the value returned by the RPC call. It is then overwritten by RPC when the Macintosh call returns, just as the Pascal VAR argument would be.

As you can see above, the Lisp function **rpc:describe-octet-structure** is often useful for examining these structures.

Defining Octet Structures

Most of the octet-structure types that you will need when writing code that interfaces to the Macintosh from Lisp are already defined for you. However, if you need to access internal data-structures in MacOS, or write interfaces to already existing programs running on the Macintosh, you may run into the need to define your own octet-structure types. The following section describes the mechanisms provided for defining octet-structures.

Forms to Define Octet Structures

rpc:define-octet-structure *name-and-options* &body *fields* *Macro*

Defines an octet structure.

name-and-options The name of the octet structure or the list containing the name of the structure and some number of keyword value pairs.

This macro takes the keyword arguments **:conc-name**, **:constructor, :default-pointer,** and **:include**, which behave in a way similar to the corresponding keywords for **defstruct**. See the section "Options for **defstruct**" for further information. In addition, **rpc:define-octet-structure** takes the following keyword arguments:

:access-type Specifies how references are made by default as one of **:octet**, **:unsigned-8**, or **:byte-swapped-8**. To define octet structures to represent Macintosh structures, for use by the Toolbox remote entries, always use **:byte-swapped-8**. The default is **:octet**.

:alignment Controls the automatic insertion of padding. This is useful when defining structures that you want to correspond directly to structures defined in another language or on a different architecture or both. **:alignment** takes an integer value as an argument. Specifying an alignment of *n* means that all structure fields of size greater than or equal to *n* should be aligned with the next offset evenly divisible by *n*. Where the field is a vector it will be aligned based on the element size of the vector.

For example, when defining octet-structures in Lisp to represent C structs as defined by THINK C on the Macintosh, you will want to specify an alignment of 2. This will align all structure fields of 2 or more bytes to an even byte boundary, and make sure that the Lisp accessors defined by **rpc:define-octet-structure** correspond precisely to where the data is stored by C. In any situation where you are using octet-structures to represent data that is created by one machine/language and manipulated by another, it is essential that you take into account the storage conventions of the other implementation.

:define-accessors Inhibits definition of macros.

:export As for **defstruct**, takes a list of keywords from the set (**:structure-name :accessors :constructor**).

:default-type An integer type. :**integer-8** is the default.

:default-index Makes the second argument to each accessor optional, defaulting it to the value you supply with this argument.

Elements of *fields* can be a symbol for single unsigned byte fields, or a list of field name and type. * used for a name allocates space, but doesn't define accessors. * used as a type defines subfields that overlap, such as bit fields. + used as a name defines unions.

Examples

These examples show the C types and corresponding octet-structure definitions for Point, Pattern, and Penstate.

Point

```
typedef struct
        {
        int     v,h;
        } Point ;

(define-octet-structure point :access-type :byte-swapped-8
                        :default-type :integer-16
  vertical
  horizontal)
```

Pattern

```
typedef unsigned char Pattern[8];

(define-octet-structure pattern :access-type :byte-swapped-8
                        :default-type :integer-16
  (data (vector unsigned-8 8)))
```

Penstate

```
typedef         struct
        {
        Point   pnLoc;
        Point   pnSize;
        int             pnMode;
        Pattern         pnPat;
        } PenState;

(define-octet-structure pen-state :access-type :byte-swapped-8
                        :default-type :integer-16
  (pen-loc point)
  (pen-size point)
  (pen-mode integer-16)
  (pen-pattern pattern))
```

An octet structure that includes elements of a C structure:

```
(define-octet-structure total
  (one one-type)
  (two two-type)
  (three three-type))
```

An octet structure that includes elements of a C union:

```
(define-octet-structure general
  (+ ((one one-type))
     ((two two-type))
     ((three three-type))))
```

The slight differences being that we do not name the subsets, just the fields, and that we always have a level of parens, instead of requiring a recursive struct when the union element has more than one field. The differences between defining octet structures that include C members and those that include C unions is that for member, fields are named, but subsets are unnamed and

Complex Example

A more complex example, which cannot be written in C or Pascal, has several variable length fields. Notice that the size of one field can depend on the contents of a different, preceding field:

```
(define-octet-structure font-header :access-type :byte-swapped-8
                        :default-type :integer-16
  (font-type cardinal-16)
  (first-char integer-16)
  (last-char integer-16)
  (width-max integer-16)
  (kern-max integer-16)
  (ndescent integer-16)
  (rect-width integer-16)
  (rect-height integer-16)
  (owt-loc integer-16)
  (ascent integer-16)
  (descent integer-16)
  (leading integer-16)
  (row-words integer-16)
  (bit-image (vector integer-16 (* rect-height row-words)))
  (location-table (vector integer-16 (+ (- last-char first-char) 3)))
  (offset-width-table (vector integer-16 (+ (- last-char first-char) 3)))
  )
```

```
(define-octet-structure font-family-header :access-type :byte-swapped-8
                       :default-type :integer-16
  (flags (* integer-16
            (image-height-p (boolean-bit 0))
            (character-width-p (boolean-bit 1))
            (fract-enable-ignore (boolean-bit 12))
            (use-extra-width (boolean-bit 13))
            (use-fract-width-table (boolean-bit 14))
            (fixed-width-p (boolean-bit 15))))
  (family-id cardinal-16)
  (first-char integer-16)
  (last-char integer-16)
  (ascent fixed-point-4+12)
  (descent fixed-point-4+12)
  (leading fixed-point-4+12)
  (width-max fixed-point-4+12)
  (width-table-offset integer-32)
  (kern-table-offset integer-32)
  (style-table-offset integer-32)
  (style-properties (vector fixed-point-4+12 9))
  (intl (vector integer-16 2))
  (version-number integer-16)
  (association-table font-family-association-table))
```

rpc:define-octet-structure-field-type *name type-arglist reference-arglist* &body *clauses* *Macro*

Defines a new octet-structure field type. Clauses are **:size, :data**, or **:expander**.

Here's a simple example from the system. OSTYPE is the Macintosh type for a string of four bytes. By providing a mechanism for retrieving four bytes from the structure at the appropriate index, and a way to alter the slot, any subsequently defined octet-structure can have a field with this type and automatically generate the appropriate **setf**able accessor.

```
(define-octet-structure-field-type ostype nil (array index)
          :size
          4
          :data
          `(ostype-from-subarray ,array ,index))

(defun ostype-from-subarray (array index)
  (let ((result (make-string 4)))
    (dotimes (j 4)
      (setf (aref result j) (code-char (aref array (+ j index)))))
    result))
```

```
(defun set-ostype-from-subarray (array index value)
  (setq value (string value))
  (check-type value (vector string-char 4))
  (dotimes (j 4) (setf (aref array (+ j index)) (char-code (aref value j))))
  value)

(defsetf ostype-from-subarray set-ostype-from-subarray)
```

Also see the section "Predefined Octet Structure Field Data Types".

rpc:define-octet-structure-field-type-macro *name arglist expansion* *Macro*

Defines an abbreviation for a field type. For example:

```
(define-octet-structure-field-type-macro OSERR () 'integer-16)
```

This defines the field-type OSERR to be an abbreviation for the system-defined field-type integer-16.

rpc:define-octet-structure-conversion-field-type *name expansion* &key *(:conversion* **'identity***) (:result-type* **'t***)* *Macro*

Defines a field type that is built on another with just a final data conversion, such as a sign extension or bit reversal.

Here's an example from the system:

```
(define-octet-structure-conversion-field-type fixed-point-16 integer-16
  :conversion fixed-point-16)

(defun fixed-point-16 (word)
  (* (ldb (byte 16 0) word) (scale-float 1.0 -16)))

(defun un-fixed-point (float)
  (round (ash float 16)))

(define-setf-method fixed-point-16 (ref)
  (let ((store (gensym)))
    (values nil nil (list store) `(setf ,ref (un-fixed-point ,store)) ref)))
```

The fixed-point-16 field is used to store a fractional number between 0 and 1. To make this convenient to manipulate in Lisp, we would want to be able to pass Lisp floating-point numbers as arguments, and have the accessor convert the float to a 16-bit integer to store in the slot. The previous example accomplishes this.

Predefined Octet Structure Field Data Types

Atomic field types

When defining octet-structures it is often useful to define a new field type in terms of some conversion routine, or some expansion, of a previously defined type. Although most of the MacOS types are already defined as octet-structure field types, the following is a short list of the basic field types: those which are most useful when defining your own octet structures and octet-structure field types.

unsigned-byte (&optional *(size 8)*)

signed-byte (*size* &)

integer-32 Equivalent to (**signed-byte 32**).

cardinal-32 Equivalent to (**unsigned-byte 32**).

integer-16 Equivalent to (**signed-byte 16**).

cardinal-16 Equivalent to (**unsigned-byte 16**).

integer-8 Equivalent to (**signed-byte 8**).

cardinal-8 Equivalent to (**unsigned-byte 8**).

padding (*base-type* &optional *(repeat 1)*)
Just occupies space; the field cannot be accessed. *base-type* can be any field type.

vector (*type length*) *length*
Can be a form that references earlier structure elements (such as repeat byte count). Referencing returns a vector of the elements, or you can use the **octet-structure-field-elements loop** iteration path.

load-byte (*base-type position size*)
Accesses a subfield of *base-type* access. Useful with *.

bit (*base-type bit-number*)
Equivalent to **(load-byte base-type bit-number 1)**.

boolean (*base-type*) Equivalent to *base-type* with not-temp test.

boolean-bit (*base-type bit-number*)
Equivalent to **(boolean (bit** *base-type bit-number***))**.

member (*base-type set*)
set is a form to evaluate to a sequence indexed by field.

subset (*base-type keywords*)
One bit for each position to indicate the presence of corresponding element.

character-8 () Equivalent to unsigned-8 with code-char input and char-code output.

ascii-character-8 () Equivalent to unsigned-8 with ASCII-char input and char-ASCII output.

Octet Structure Accessor Forms

rpc:octet-structure-field-ref (*structure-name field-name array index*)
References a field.

rpc:octet-structure-field-index (*reference*)
Returns index in array of start of this field. *reference* is of the form (conc'ed-name array index).

rpc:octet-structure-field-size (*reference*)
Returns size of this field in bytes.

rpc:octet-structure-total-size (*reference*)
Returns size of whole structure in bytes; that is, the index after the last field. *reference* is of the form (structure-accessor array index).

rpc:octet-structure-field-entry (*reference*)
References a field pointed to directly by the index, rather than to the head of the whole structure.

rpc:with-octet-structure-access-type (*type &body body*)
Overrides the default referencing mode, for using the same definition for more than one data representation (such as Symbolics C and foreign system binary files).

rpc:with-octet-structure-fields ((*structure-name* &rest *reference-args) fields* &body *body*)
Binds lexical variables to structure slots. Elements of *fields* either a symbol or a list of local variable and field name.

loop iteration over octet-structure-field-elements

loop for *index* being the **rpc:octet-structure-field-elements** of *reference* Map *index* over the the successive elements of a vector field. Presumably the field type is another structure type, whose accessors you then use to process each element.

Octet Structure Field Operations

rpc:copy-octet-structure (*to-reference from-reference*)
Replaces the contents of the octet-structure referred to by to-reference with the contents of from-reference.

rpc:octet-structure-equal (*reference-1 reference-2*)
Does a byte by byte comparison of the octet-structures referred to by reference-1 and reference-2.

Octet Structure Flavor and Structure Synchronization

rpc:define-octet-structure-and-flavor (*structure-name inherit-from* &body *fields*)
Defines a flavor and structure that parallel one another. The flavor has instance variables with same names as field elements.

rpc:copy-fields-from-octet-structure (*instance array index*)
Copies from the array into the instance, and returns the index past the end of the structure. Uses appropriate method combination to get *inherit-from*'s fields too.

Octet Structure Debugging

rpc:describe-octet-structure (*structure-name array index*)
Describes the contents of each field in the octet-structure referenced by *structure-name array* and *index*.

MacIvory Extensions to the Macintosh Toolbox

Window Structures

mtb:_windowstructure *window-pointer window* *Function*

Returns the contents of the window structure pointed to by the Macintosh pointer window-pointer in the Lisp structure cwindow. It is particularly useful when trying to examine the state a Macintosh window from Lisp. The *window* argument is overwritten to contain the new information and returned.

mtb:_cwindowstructure *cwindow-pointer cwindow* *Function*

Returns the contents of the color window structure pointed to by *cwindow-pointer* in the Lisp structure *cwindow*. Like **mtb:_windowstructure**, it is useful for examining the state of Macintosh windows, and overwrites and returns the argument *cwindow*.

Access to Macintosh Memory

mtb:_write-opaque-bytes-into-pointer *ptr nbytes buffer* *Function*

Writes the first *nbytes* of buffer into the Macintosh memory at the location specified by *ptr*. *Ptr* should be a Macintosh pointer (as returned by _newptr), *buffer* should be an 8-bit array. Memory on the Macintosh should be allocated prior to calling this function.

mtb:_read-opaque-bytes-from-pointer *ptr nbytes buffer* *Function*

Reads *nbytes* from the Macintosh memory pointed to by *ptr* into the Lisp array designated by *buffer*. The argument *buffer* is overwritten and returned.

mtb:_write-opaque-bytes-into-handle *h nbytes buffer* *Function*

Writes the first *nbytes* of *buffer* into the Macintosh memory at the location specified by *h*. The memory on the Macintosh should be allocated prior to calling this function.

mtb:_read-opaque-bytes-from-handle *h nbytes buffer* *Function*

Reads *nbytes* from the Macintosh memory pointed to by *h* into the Lisp array designated by *buffer*. The argument *buffer* is overwritten and returned.

Miscellaneous

mtb:_copybits *srcbits dstbits srcrect dstrect mode maskrgn* &key *:transport-agent* *Function*

mtb:_drawpixmap *srcbits srcrect dstrect mode maskrgn* *Function*

Draws the pixMap *srcbits* on the current GrafPort's portBits. Other arguments are as for **mtb:_copybits**.

mtb:_fsread-remote *refnum into len* *Function*

On the Macintosh side, reads *len* bytes from the file open on Macintosh refnum *refnum*, into the Macintosh storage at the Ptr *into*.

mtb:_fswrite-remote *refnum from len* *Function*

On the Macintosh side, writes *len* bytes from the Ptr *from* into the file open on Macintosh refnum *refnum*.

mtb:_ptrfromhandle *h* *Function*

Given a handle *h*, returns the pointer to the contents of the handle. As with all handle dereferencing on the Macintosh, you should lock the handle before retrieving the pointer to its contents and be sure to unlock the handle as soon as possible to keep memory from becoming fragmented.

mtb:_setcursorfromhandle *crsrhandle* *Function*

Performs a Setcursor on the result. This function is provided as a convenience because cursors are usually stored as resources that are accessed by handles.

mtb:_unpackbitsbyrows *srcptrin dstptrin rowbytes nrows* *Function*

Calls UnpackBits starting with *srcptrin*, *dstptrin*, and *rowbytes*, for *nrows* times. Returns two values, the final values taken on by the src and dst Ptrs.

MacIvory Toolbox Macros

mtb:cconstant-case *test-object* &body *clauses* *Macro*

Provides a case statement with keys that are determined by looking up the value of the Macintosh constant. Like **case**, **mtb:cconstant-case** allows an otherwise clause. Here is a simple example:

```
(defun my-stdrect (graphic-operation rect)
  (cconstant-case graphic-operation
    (frame (my-frame-rect rect))
    (paint (my-paint-rect rect))
    (erase (my-erase-rect rect))
    (invert (my-invert-rect rect))
    (fill (my-fill-rect rect))))
```

Note that `frame`, `paint`, `erase`, `invert`, and `fill` are defined as constants on the Macintosh (see *Inside Macintosh*, volume I), and are exported from the **mtb** package in Genera.

mtb:cconstant-ecase *test-object* &body *clauses* *Macro*

Like **mtb:cconstant-case**, but signals an appropriate error if none of the constants match *test-object*. See the macro **mtb:cconstant-case** for further information.

mtb:with-qd-port *(new-port)* &body *body* *Macro*

Saves the current Quickdraw port, as found by calling **mtb:_getport**, executes the body with the current port set to *new-port*, and restores the original port when completed. This is convenient for doing output to Macintosh windows.

mtb:with-handle-locked *(handle* &optional *(lock* **'t**)) &body *body* *Macro*

Executes *body* with handle locked. You can also pass in a computed boolean, which if false means don't bother to lock. Macintosh handles must be locked before they can be dereferenced. This prevents the Macintosh memory-manager from relocating the memory referred to by the handle.

See *Inside Macintosh*, volume II, for more information on memory management.

mtb:with-resource-handle *(var restype resid-or-name* &key *:lock :file :sole-file (:direction* **:io**)) &body *body* *Macro*

Causes the Macintosh to read a specified resource by calling `_getresource` or a similar function on *restype* and *resid-or-name*. It binds a Lisp *var* to the handle to that resource. *resid-or-name* is an integer resource-id or string, and is used to find the resource.

If **:lock** is non-**nil** the resource is locked around the body. If **:file** is provided, it is opened and used as a resource file around the body. If **:sole-file** is provided, **mtb:with-resource-handle** looks only within that file for the resource. The **:sole-file** keyword takes a file reference number (use this instead of a pathname if the file is already open).

Values for **:direction** are **:input**, **:output**, or **:io**; **:io** is the default and is synonymous with **:output**. **:input** means open the file in read-only mode.

mtb:with-resource *(var restype resid-or-name* &key *:lock :file :sole-file (:direction* **:io**)) &body *body* *Macro*

Similar in function to **mtb:with-resource-handle**, but copies the resource contents to Lisp and binds the Lisp variable to the structure. This is useful for writing Lisp programs that read or modify Macintosh resources.

See the macro **mtb:with-resource-handle** for further information.

mtb:with-mac-struct *(lisp-var octet-structure-type* &rest *options* &key *:handle :ptr)* &body *body* *Macro*

Copies a structure from the Macintosh system and binds a Lisp variable to it.

Octet-structure-type (not evaluated) determines the structure type. You provide the location of the structure on the Macintosh by specifying either **:ptr** or **:handle**. When the structure type is specified as **nil**, raw bytes, are copied, with size determined from the size of the object on the Macintosh side. This macro is useful for writing Lisp programs that manipulate Macintosh data structures.

mtb:with-mac-temp *(lisp-var* &rest *options* &key *:handle :ptr :size :initial-contents)* &body *body* *Macro*

Allocates a data structure on the Macintosh for the duration of *body*, binding a Lisp variable to its `Ptr` or `Handle`. Its size is 0, or *size*, or length of **:initial-contents**, which is downloaded into the **handle** or **ptr**. The memory is allocated on the Macintosh and deallocated on exiting the body. Unlike **mtb:with-mac-struct**, **mtb:with-mac-temp** does not copy the contents of the Macintosh memory to the Lisp side.

mtb:with-resload *(new)* &body *body* *Macro*

Binds the state of ResLoad to *new*, restoring the old state on exiting the body.

mtb:using-resfile *(refnum)* &body *body* *Macro*

Uses **mtb:_useresfile** of *refnum* around the body, restoring the previous value of currentresfile on exit.

mtb:with-resfiles *((var name &optional direction) ...)* *Macro*

Opens one or more resource files for *body*, binding Lisp variables to their refnums. *Direction* is **:input** (the default), or one of the synonyms **:output** or **:io**. The resource files are closed on exit.

mtb:do-restypes *(type-var refnum-or-all)* &body *body* *Macro*

Iterates over all resource types in one resource file, specified by *refnum-or-all*, binding *type-var* to each type. When *refnum-or-all* is **:all**, **mtb:do-restypes** iterates over all types in all open resource files.

mtb:do-rsrcs *(handle-var id-var name-var) (restype refnum-or-all* &key *:load)* &body *body* *Macro*

Iterates over resources of a given type, receiving a resource handle, id, and name. *refnum-or-all* is as in **mtb:do-restypes**. When **:load** is **t**, **mtb:do-rsrcs** loads each resource.

mtb:map-over-mac-queue-elements *(element-var octet-structure-type queue-hdr-pointer)* &body *body* *Macro*

Executes *body* once for every element in a Macintosh system queue. On each iteration *element-var* is bound to a Lisp octet-structure of type *octet-structure-type* containing a copy of the Macintosh queue-element.

queue-hdr-pointer is the pointer to the queue-hdr structure in Macintosh memory (as returned by **mtb:_getvcbqhdr**, for example). This macro is useful for mapping over the Macintosh operating system queues, such as the vcb queue, the drive queue, or the vertical retrace queue.

mtb:with-open-refnum *(var pathn* &key *:vrefnum :dirid :permission :resource-fork)* &body *body* *Macro*

Opens the host file named by the Lisp pathname *pathn* for the duration of *body*, binding *var* to the resulting refnum. Used when a MacOS refnum is needed instead of a Lisp stream (in contrast to **with-open-file**). The **:permission**, **:dirid**, and **:vrefnum** arguments are passed to the Macintosh File Manager (**mtb:_phbopen**) to open the file. If **:resource-fork** (default **nil**) is true, a stream to the file's resource fork is opened instead of the data fork. (PBHOpenRF vs. PBHOpen). If a Macin-

tosh operating system error occurs while attempting to open the file, an appropriate Lisp condition is signaled.

mtb:with-pict-from-file *(handle-var rect &key pathname refnum (temporary T) &body body* *Macro*

Allocates a Macintosh handle to receive the contents of a PICT file, reads the file into the handle, and binds *handle-var* to the pict handle and *rect* to its picframe, for the duration of *body*. Specify the file with the Lisp pathname *pathname*, or provide an already open MacOS refnum in *refnum*. If you specify **:temporary nil**, the Macintosh handle is not deallocated at the end of *body*.

mtb:show-resfiles *Function*

Shows the linked list of open MacOS resource files, offering the refnum and file name. It attempts to mark the current resource file with "*→".

mtb:ez-sfgetfile *&key x-and-y types* *Function*

Interface to **mtb:_sfgetfile**. *x-and-y* is a list (*x y*) or a Point, defaulting to 100,100. *types* determines what types of files are offered by SFGetFile. Returns three values: *vrefnum*, *fname* (string), and *ftype* (string). If no **:types** argument is supplied, then any file type is allowed.

mtb:octet-ref-remote-ptr *(octet-structure-field-accessor ptr &optional offset)* *Macro*

Remotely performs the specified octet structure field access by reading just the necessary bytes from the Macintosh location specified by *ptr* and *offset*. **setf** can be used with **mtb:octet-ref-remote-ptr** expressions.

Examples:

- The common operation of finding the QuickDraw port rectangle of a Macintosh window can be done with:

  ```
  (octet-ref-remote-ptr (grafport-portrect (_frontwindow))
  ```

  This is equivalent to, but faster than, transferring the entire window structure into Lisp:

  ```
  (with-mac-struct (window windowrecord :ptr (_frontwindow))
    (grafport-portrect window 0))
  ```

- You could increment the refcon of a Macintosh window with:

  ```
  (incf (octet-ref-remote-ptr (windowrecord-refcon mac-window)))
  ```

- To move the bounds rectangle for a dialog template whose Macintosh ptr is in the Lisp variable `the-template`, you can use:

```
(setf (octet-ref-remote-ptr
          (dialogtemplate-boundsrect the-template))
      (_offsetrect (octet-ref-remote-ptr
                        (dialogtemplate-boundsrect the-template)) 10 0))
```

mtb:octet-ref-remote-handle *(octet-structure-field-accessor handle &optional offset)* *Macro*

Like **mtb:octet-ref-remote-ptr**, except this is applied to a handle. The handle is locked around the access. *offset* applies to the dereferenced *handle*. **setf** can be used with **mtb:octet-ref-remote-handle** forms.

For example, given a ptr to a Macintosh window, you could collect a list of all of its controls with their refcons:

```
(loop for control = (octet-ref-remote-ptr
                          (windowrecord-controllist mac-window))
                  then (octet-ref-remote-handle
                          (controlrecord-nextcontrol control))
      until (zerop control)
      collect (let ((refcon (octet-ref-remote-handle
                                 (controlrecord-contrlrfcon control))))
                (list control refcon)))
```

mtb:write-remote-number *&key byte word long ptr handle offset* *Function*

Writes the specified byte, word, or long into the specified Macintosh *ptr* or *handle*, at a location offset by *offset* if supplied. Useful for callbacks altering Pascal **var** parameters.

mtb:with-rect *(var &optional left top right bottom) &body body* *Macro*

Binds *var* to a data-stack Rect for the duration of *body*. Unspecified edges default to zero. Useful when dealing with Macintosh toolbox routines.

For example, one could draw a wide rectangle on the current GrafPort with

```
(with-rect (r 10 10 100 30) (_framerect r))
```

As another example, one could move a dialog item with this:

```
(with-rect (r)
  (multiple-value-bind (item-type item-handle)
        (_getditem the-dialog item-number r)
    (_offsetrect r 20 0)
    (_setditem the-dialog item-number item-type item-handle r)))
```

mtb:with-point *(var x y) &body body* *Macro*

Binds *var* to a data-stack Point for the duration of *body*. Useful when dealing with Macintosh toolbox routines.

mtb:describe-eventrecord *event-record* *Function*

Describes an octet-structure containing an EventRecord, decoding the event kind, mouse button state. Prints to ***standard-output***. Probably useful only for debugging.

mtb:with-temps *(&rest var-creator-destructor-triples) &body body* *Macro*

Binds one or more Lisp variables to freshly created entities, for the duration of *body*, destroying them (if non-null) when *body* terminates. The advantage of this over unwind-protect is that it places the creator and destroyer syntactically adjacent. Each *creator* is a function applied to no arguments (that is, **mtb:with-temps** wraps a set of parentheses around it); each *destructor* is a function applied to the corresponding *var*.

```
(with-temps ((update-region _newrgn _disposergn)
             (ptr (lambda () (_newptr size)) _disposptr))
  ---)
```

mtb:mac-pathname-from-parts *vrefnum dirid name* *Function*

Given pieces of a Macintosh pathname, constructs a Lisp pathname. Recursively looks up names of parent folders until reaching the file system root.

Example:

```
(defun std-file-accept-pathname ()
  (multiple-value-bind (vrefnum name type)
      (mtb:ez-sfgetfile :types '("TEXT" "PICT"))
    (values (mtb::mac-pathname-from-parts vrefnum 0 name) type)))
```

mtb:with-mac-file-struct *(var octet-structure-type filepos refnum)* &body *body* *Macro*

Binds *var* to an octet structure of the named type read at *filepos* from the Macintosh file open on refnum.

Example:

```
(defun size-of-pict-in-pict-file (pathname)
  (declare (values width height))
  (mtb:with-open-refnum (in pathname :permission (mtb:cconstant fsRdPerm))
    (mtb::with-mac-file-struct (pict mtb::picture 512 in)
      (let ((rect (mtb:picture-picframe pict 0)))
        (values (- (mtb:rect-right rect 0) (mtb:rect-left rect 0))
                (- (mtb:rect-bottom rect 0) (mtb:rect-top rect 0)))))))
```

mtb:with-mac-file-bytes *(var len filepos refnum)* &body *body* *Macro*

Binds *var* to an octet vector of length *len*, read at filepos from the Macintosh file open on refnum.

mtb:do-mac-volumes *(volume-name volume-refnum* &optional *volumeparam)* &body *body* *Macro*

Iterates over all mounted MacFS volume structures. Body receives the *volume-name* string (without trailing colon), that volume's refnum, and, if needed, a VolumeParam structure. The string containing the volume name, and the VolumeParam structure, are allocated on the data stack, so they must be copied if they are to be used later.

Example:

```
(mtb::do-mac-volumes (name refnum) (print (list name refnum))) ==>
   ("Mac-4" -1)
   ("DSK" -2)
   ("SneakerNet Packet Floppy" -3)
```

mtb:with-one-mac-rsrc *(var pathname rsrc-type* &key *:rsrc-id :rsrc-name (:vrefnum* **0***) (:dirid* **0***))* &body *body* *Macro*

Finds a resource of type *rsrc-type,* id **:rsrc-id**, or name **:rsrc-name** in the Macintosh file *pathname* (adjusted by *vrefnum* and *dirid*), moves it to the Lisp side, and binds *var* to it. **mtb:with-one-mac-rsrc** also decodes the resource file. For resource files with large resource maps, this offers increased performance over using **mtb:with-resfiles** or **mtb:with-resource :sole-file** with Macintosh's Resource Manager.

For related information, see "The Resource Manager is not a database." -- Macintosh Technical Note #203: *Don't Abuse the Managers*, August 1, 1988.

mtb:_debugger &key *:transport-agent* *Function*

Enters the Macintosh debugger (Macsbug, for example), after halting the Macintosh at an instruction in the RPC remote entry.

mtb:_debugstr *string* &key *:transport-agent* *Function*

Enters the Macintosh debugger (Macsbug, for example), after halting the Macintosh at an instruction in the RPC remote entry, passing it *string* to print as part of its greeting.

mtb:using-scratch-resfile *()* &body *body* *Macro*

Runs *body* using a scratch resource file as the current resource file. The scratch file is deleted when you quit the Genera icon.

mtb:download-resource *res-type res-name res-id res-attr data* *Function*

Creates a Macintosh resource in a scratch resource file. The first four arguments define the resource. *data* is the contents of the resource. **mtb:download-resource** returns two values: a handle and *res-id*. When you specify **nil** as *res-id*'s value, **mtb:download-resource** generates a unique id for the resource.

Callbacks

A *callback* is a piece of user code whose address is given to MacOS to be called by MacOS at some later time to perform some specific function defined by MacOS. Examples of callbacks are: custom QuickDraw QDProcs, an actionProc passed to TrackControl, or a filterProc passed to ModalDialog. The interfaces described here can construct temporary Macintosh routines (essentially closures on the Macintosh side) to serve as "trampolines" to pass callback args to Lisp, and to receive Lisp's values to pass back to MacOS. A more complete technical description of the implementation appears in "Callback Mechanism" . Because each kind of callback takes different arguments and returns different values (or no values), we use *callback types*, which are symbolically named.

In general, callbacks are useful only in the context of a remote-program. Each callback call is processed in a new process spawned by the RPC machinery.

Callbacks receive args from MacOS, and can return values if MacOS specifies them as functions. Pascal **var** parameters, passed to callbacks as Ptrs, can be altered with **mtb:write-remote-number**.

The callback constructor functions in the **mtb** package return a Macintosh ProcPtr, represented in Lisp as a fixnum. Such an address is ready to be installed into Macintosh data structures, passed back to some Macintosh Toolbox routine, or whatever else is required by the application.

Each callback establishes a unique association with the specified Lisp code. This admits many co-extant callbacks of the same type without conflict or confusion.

mtb:with-mac-callback *(var callback-type &key program) ((handler-args) &body handler-body) &body body* *Macro*

For the duration of *body*, binds *var* to the Macintosh address for a callback of type *callback-type*. When the callback is called, bind *handler-args* to its args, and run *handler-body* in its own process. The callback's return value (if needed) is the value of *handler-body*. *program* is a remote program, defaulting to **dw:*program***. Although *handler-body* is run in another process, it does, of course, share the lexical environment in which it appears.

mtb:with-mac-callback-function *(var callback-type function &key program)* *Macro*

For the duration of *body*, binds *var* to the Macintosh address for a callback of type *callback-type*. When the callback is called, apply *function* the callback's args. The callback's return value (if needed) is the value returned by *function*. *program* is a remote-program, defaulting to **dw:*program***.

mtb:make-mac-callback *callback-type function program* *Function*

Constructs a callback "closure" on the Macintosh side, of callback-type *callback-type*, which when called will call the Lisp *function* in the context of the remote program *program*. Returns the Macintosh address of the callback This is undone by **mtb:remove-mac-callback**.

mtb:remove-mac-callback *mac-callback* *Function*

Deallocates the callback addressed by *mac-callback* and removes the Lisp data structures establishing the correspondence with the Lisp function.

Callback Types

This section describes the arguments and values for each callback type, along with associated C types. Call-by-reference parameters (Pascal **var** parameters) are noted with C syntax.

mtb:control-definition *Callback*

Arguments:
varcode `short`
thecontrol `ControlHandle`
message `short`
param `long`
Values:
result `long`

Custom control defintion. See *Inside Macintosh*, page I-329. Because of the Macintosh's Control Manager design, you will have to adjust CDEF resources before you can actually get one of these called.

mtb:control-indicator-action *Callback*

Arguments: *none*
Values: *none*

A control's action procedure for when the mouse was pressed in an indicator. See *Inside Macintosh*, page I-324. Suitable for passing to _TrackControl or _SetCtlAction.

mtb:control-nonindicator-action *Callback*

Arguments:
thecontrol `ControlHandle`
partcode `short`
Values: *none*

A control's action procedure for when the mouse was pressed in some control other than an indicator. See *Inside Macintosh*, page I-324. Suitable for passing to **mtb:_trackcontrol** or **mtb:_setctlaction**.

mtb:dialog-sound-proc *Callback*

Arguments:
soundno `short`
Values: *none*

Emits alert sounds. See *Inside Macintosh*, page I-409.

mtb:dialog-user-item-definition *Callback*

Arguments:
thewindow `WindowPtr`
itemno `short`
Values: *none*

Draws userItems in dialogs. See *Inside Macintosh*, page I-421-2 and I-431.

mtb:menu-definition *Callback*

Arguments:
message `short`
themenu `MenuHandle`
menurect `Rect*`
hitpt `Point*`
whichitem `short*`
Values: *none*

Implements custom menus. See *Inside Macintosh*, page I-362. Because of the Macintosh's Menu Manager design, you must use MDEF resources to get one of these to be called.

mtb:modal-dialog-filter *Callback*

Arguments:
thedialog `DialogPtr`
theevent `EventRecord*`
itemhit `short*`
Values:
result `Boolean`

Preprocesses events for ModalDialog. See *Inside Macintosh*, page I-415.

mtb:quickdraw-arcproc *Callback*

Arguments:
verb `GrafVerb`
r `Rect*`
startangle `short`
arcangle `short`
Values: *none*

Customizes QuickDraw arc/wedge drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-bitsproc *Callback*

Arguments:
srcbits `BitMap*`
srcrect `Rect*`
dstrect `Rect*`
mode `short`
maskrgn `RgnHandle`
Values: *none*

Customizes QuickDraw bit transfer. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-commentproc *Callback*

Arguments:
kind `short`
datasize `short`
datahandle `Handle`
Values: *none*

Customizes QuickDraw picture comment processing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-getpicproc *Callback*

Arguments:
dataptr `Ptr`
bytecount `short`
Values: *none*

Customizes QuickDraw picture retrieval. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-lineproc *Callback*

Arguments:
newpt `Point*`
Values: *none*

Customizes QuickDraw line drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-ovalproc *Callback*

Arguments:
verb `GrafVerb`
r `Rect*`
Values: *none*

Customizes QuickDraw oval drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-polyproc *Callback*

Arguments:
verb `GrafVerb`
poly `PolyHandle`
Values: *none*

Customizes QuickDraw polygon drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-putpicproc *Callback*

Arguments:
dataptr `Ptr`
bytecount `short`
Values: *none*

Customizes QuickDraw picture saving. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-rectproc *Callback*

Arguments:
verb `GrafVerb`
r `Rect*`
Values: *none*

Customizes QuickDraw rectangle drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-rgnproc *Callback*

Arguments:
verb `GrafVerb`
rgn `RgnHandle`
Values: *none*

Customizes QuickDraw region drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-rrectproc *Callback*

Arguments:
verb `GrafVerb`
r `Rect*`
ovalwidth `short`
ovalheight `short`
Values: *none*

Customizes QuickDraw roundRect drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-textproc *Callback*

Arguments:
bytecount `short`
textbuf `Ptr`
numer `Point*`
denom `Point*`
Values: *none*

Customizes QuickDraw text drawing. See *Inside Macintosh*, page I-197ff.

mtb:quickdraw-txmeasproc *Callback*

Arguments:
bytecount `short`
textaddr `Ptr`
numer `Point*`
denom `Point*`
info `FontInfo*`
Values:
width `short`

Customizes QuickDraw text width measurement. See *Inside Macintosh*, page I-197ff.

mtb:standard-file-dialog-hook *Callback*

Arguments:
item `short`
thedialog `DialogPtr`
Values:
result `short`

Customizes StandardFile dialogs. See *Inside Macintosh*, page 522.

mtb:standard-file-file-filter *Callback*

Arguments:
paramblock `ParmBlkPtr`
Values:
result `Boolean`

Filters out file types for StandardFile dialogs. See *Inside Macintosh*, page I-524.

mtb:text-edit-click-proc *Callback*

Arguments: *none*
Values:
result `Boolean`

Customizes TextEdit clikLoop. See *Inside Macintosh*, page I-390.

mtb:text-edit-word-break-proc *Callback*

Arguments:
text `Ptr`
charpos `short`
Values:
result `Boolean`

Customizes TextEdit word break computation. See *Inside Macintosh*, page I-390.

mtb:window-definition *Callback*

Arguments:
varcode `short`
thewindow `WindowPtr`
message `short`
param `long`
Values:
result `long`

Implements custom windows. See *Inside Macintosh*, page I-299.

mtb:get-canned-callback *canned-callback-type* *Function*

Returns the Macintosh address of a simple, parameterless callback. Some callbacks are so simple that they can be written in C, to run entirely on the Macintosh side, instead of communicating back to the Lisp side. At present, there is only one canned callback type, **mtb:draw-dialog-line**.

Callback Mechanism

This section briefly describes how the callback mechanism is implemented.

The call chain when MacOS calls a callback is this:

mac-closure
→ C-client remote entry
→ Lisp-server remote entry
→ Lisp callback function

Here *mac-closure* is a sequence of M68000 machine instructions constructed by Lisp and sent over to the Macintosh, containing as immediate data five variable fields: a closure id, application id, value to load register `A4` with, the address of the C-client remote entry, and the number of bytes of arguments to clear off the stack. Here is the assembly code. The variable fields to be filled in by Lisp are marked in bold.

```
link a6,#-4
move.l a4,-(sp)
movea.l #a4value,a4

;; push C args, right to left
pea -4(a6)                    ;location for return value, if needed
pea 8(a6)                     ;addr of last (Pascal) arg
pea appl_id                   ;application unique-id
pea closure_id                ;closure code (not address..)
```

```
jsr @#xxxx                              ;call the RPC remote entry
lea 16(a7),a7                           ;C caller must remove args
;; Fail to check D0 for RPC errors -- what could we do anyway?
move.l -4(a6),d0                        ;return value

move.l (sp)+,a4                         ;restore old A4
unlk a6
movea.l (a7)+,a0                        ;return addr
lea argsize(a7),a7                      ;Pascal callee must remove args
move.l d0,(a7)                          ;return value
jmp (a0)                                ;return
```

Each callback type has a C-client remote entry. Its arguments are *closure-id*, *application-id*, and an anonymous `struct*` argument representing its caller's arglist. That remote entry passes *closure-id*, *application-id*, and each arg of the caller's arglist, across the RPC mechanism to the Lisp-server side.

There is also a C-server remote entry that receives a callback-type index from Lisp-client; it returns to Lisp two longs, the Macintosh address of the specified C-client remote entry of the previous paragraph, and the current value of register `A4`.

All C-side remote entries of the previous two paragraphs are linked into a single code resource of RPC module type autoload-with-static-data, locked in Macintosh memory when loaded. Because this is an autoload code resource, it needs a correct value in register `A4` to refer to its globals. Actually, the only global is `DefaultXDRAgent`, needed by the C-client remote entries. `DefaultXDRAgent` is filled in by the callback-address-returning C-server remote entry.

All the Lisp-server sides of the callback remote entries have similar bodies. Each body will use the application-id to locate the appropriate remote program, and the closure-id to locate the appropriate Lisp function to call back. Then it applies the Lisp function to the reverse of the vector of RPC arguments received from the Macintosh side. It has to reverse the vector because the Pascal calling sequence (used by the caller of the callback in the first place) pushes the arguments left-to-right on a stack which grows toward lower addresses. Thus, the last Pascal argument appears first in memory.

Examples of Developing MacIvory User Interfaces

You can find several examples of code that creates MacIvory user interfaces in the directory SYS:EMBEDDING;MACIVORY;TOOLBOX;EXAMPLES;. These examples are printed in the appendix "MacIvory User Interface Examples". They include etest.lisp, pic-show.lisp, menubar.lisp, and show-icons.lisp.

Etest and Pic-show are remote programs. To run them, use the unassigned Genera application icon in the Macintosh Applications folder. Copy the icon and then name it with the name of the remote program. Opening the icon runs the program. (You can also use the Configure MacIvory Application command as an alternative to this procedure).

The following list describes each program:

- **etest.lisp**, which defines remote program "ETest" with three menu commands:

  ° *Test-1* (clover-S), which draws a solid black circle. You can use the scroll bars, and reshape the window.

  ° *Splash* (clover-F), which draws the Bear•Cal splash window. It reads Macintosh PICT contents from the sys host, ships it over to the Mac, writes it into a new pict, and draws that pict when the window's displayer wants it.

  ° *Dialog* (clover-D), which creates a modal dialog for an airline reservation system (Bear•Cal). It shows how to use the dialog functionality.

  credit-card-picts.rsrc and **splash.pict** are binary data files used by the ETest commands. splash.pict is taken from the sys host, so you can try it out without further work. To use the credit-card-picts.rsrc demo, you must copy it onto your Macintosh before running the Dialog demo, and then edit a **defvar** in etest to say where the file has been copied.

- **pic-show.lisp**, which contains an application that browses for Macintosh files containing pictures, and lets you see them. You can use scrolling and zooming with these windows.

- **menubar.lisp**, which demos advanced octet-structure definitions and usage, using the Macintosh menu bar as an example.

- **show-icons.lisp**, which uses toolbox macros and generic graphics substrate to show you all the icons and mouse cursors in the system file.

Launching Macintosh Applications

The following functions are provided for launching Macintosh applications from Genera:

mtb:_launch *name fdflags launchflags* &key *:transport-agent* *Function*

Launches the application in the file named by *name*, using *fdflags* and *launchFlags*.

Because of limitations of Macintosh's _launch, most MacIvory users will want to use **mtb:launch-mac-application**, which is more comprehensive, instead of the **mtb:_launch** function. See the Apple Macintosh documentation on launching applications for further information.

mtb:launch-mac-application *application-pathname* &rest *document-pathnames*
Function

Launches the Macintosh application in *application-pathname*, telling it to open the documents in *document-pathnames*. If *application-pathname* is **nil**, locates the responsible application by looking up the file types of document-pathnames in the Finder's desktop file.

The returned value is a Macintosh process-id, used, for example, to inform a shell that a child has died. See Macintosh Technical Note #205 for further information on this topic.

Example:

```
(mtb::launch-mac-application
  () #p"HOST:DSK:MacIvory Applications:HyperIvory Stack")
```

Notes: If the launch does not succeed for some reason (file not found, not enough memory), the error code returned by _launch itself (and signalled by **mtb:launch-mac-application**) does not always appear to be meaningful.

In Macintosh System 7.0, Apple plans to reimplement the Macintosh software used to launch applications. This function may be made obsolete by those changes.

Hardware-dependent Data Formats

In using Ivory embedded in standard systems (such as Macintosh) or standard busses (such as VMEbus) you sometimes have to deal with hardware that uses data formats that differ from Ivory formats; for instance, bits may be numbered in the other direction. Symbolics provides hardware that speeds up any bit shuffling this may require.

The bit-shuffling hardware is necessarily different in each system. Also, some systems do not need bit shuffling at all (for instance, the IBM PC uses the same data formats as Ivory for numbers and strings). The macro **sys:with-hardware-bit-shuffling** provides a general mechanism to cover up all this system-dependent complexity. It expands into code that works on all Ivory-based systems, by means of dispatches on **sys:*system-type**. Note that this macro is strictly for accessing host hardware. You cannot use it for general purpose bit or byte reversal, such as when decoding a file created by a foreign system, because the bit-shuffling hardware does not work that way.

sys:with-hardware-bit-shuffling *(variable mode) body* *Macro*

The *variable* contains a pointer to a block of storage that is to be accessed with bit shuffling in effect during the execution of the *body*. Bit shuffling is per-process (actually per-stack-group). The block of storage must be in a portion of address space that is capable of using the bit shuffling hardware. In most systems this requires dtp-physical-address and is not the portion of physical address space that is used for storing virtual pages. Thus when doing a bit shuffling block transfer between an I/O device and main memory, *variable* points at the I/O device. If *variable* doesn't meet these requirements, an error is signaled. In some systems, such as MacIvory, bit shuffling is (partially) encoded in the address bits; for this reason, a

new variable named *variable* is bound to a potentially modified version of the address in *variable*; its scope includes *body*.

The *mode* identifies the units of storage to remain intact through the bit shuffling. It must evaluate to one of the following keyword symbols:

:bit	Single bits, **dtp-fixnum** tag
:nibble	4 bits at a time, **dtp-fixnum** tag
:byte	8 bits at a time, **dtp-fixnum** tag
:fixnum	32 bits at a time, **dtp-fixnum** tag
:single-float	IEEE 32-bit float, **dtp-single-float** tag

If the host hardware uses the same data format as the Ivory for these units, no shuffling occurs. Otherwise, data words are rearranged as they are read or written to translate the formats. For example, **:bit** mode in MacIvory reverses the order of bits in a word, to match Ivory's least-significant-bit-first order to the Macintosh's most-significant-bit-first order.

Enhancing MacIvory Performance

Using Off-Screen Bitmaps

Off-Screen Bitmaps

The Genera window system stores images in *bitmaps*: two-dimensional arrays of bits. Sometimes these are called *pixmaps* (two-dimensional arrays of pixels), especially when there is more than one bit per pixel, as on color screens. Each bitmap has a *width*, a *height*, and a *depth*; the depth is the number of bits per pixel.

A bitmap can be on the screen or off the screen. An on-screen bitmap is stored in special memory associated with the display hardware, thus its contents is visible on the screen. The image of an exposed window resides in an on-screen bitmap so you can see it.

All other bitmaps are off-screen. The Genera window system often uses an off-screen bitmap to save the image of a deexposed window, so that the image can be quickly restored when the window is exposed, by copying the off-screen bitmap into an on-screen bitmap. This feature is under the control of the individual window, through the **:save-bits** option, however most standard windows enable it. The Genera window system also uses an off-screen bitmap to save the image underneath a temporary window, such as a pop-up menu. User programs can also use off-screen bitmaps for their own purposes, using the facilities described below.

In embedded systems, such as MacIvory, bitmaps have one more degree of freedom: a bitmap can be in Ivory's normal virtual memory, or it can be in host memory. On-screen bitmaps are always in host memory, because the host owns the dis-

play hardware. This is why the screen-array bitmap of an exposed window generally cannot be directly accessed by an Ivory program in an embedded system. Similar considerations apply to X Windows, substituting X Window Server memory for host memory.

Off-screen bitmaps can be in either Ivory memory or host memory. There are performance tradeoffs associated with the decision of where to store an off-screen bitmap. The principal constraints are the limited speed for copying images between Ivory memory and host memory, and the limited size of host memory.

On the one hand, placing an off-screen bitmap in host memory yields the maximum speed when copying images to or from the screen. Making the host Macintosh solely responsible for the copying is significantly faster than having to coordinate two processors and copy through the co-processor communications memory. You will not see the screen being slowly repainted top to bottom.

On the other hand, placing an off-screen bitmap in Ivory memory avoids consuming the limited amount of available host memory. In a minimum-configuration Macintosh II, there is only 1 megabyte of host memory, of which about 350 to 400 kilobytes is available for off-screen bitmaps. In a larger Macintosh II, up to a few megabytes might be available, but even that may hold only a few large bitmaps. The size of a bitmap depends on the application and on the size and depth of the display monitor.

To avoid severely limiting the number of off-screen bitmaps, MacIvory swaps less recently used bitmaps onto the disk to make room in host memory. Of course, since swapping to and from the disk takes time, you will benefit from having as many bitmaps as possible in memory at once. This is controlled by the amount of host memory available to the Genera application. Since the Macintosh does not have a flexible multiple-application memory allocation scheme, this is controlled by the size set for the Genera application with the Get Info command in the Finder's File menu, and limited by the amount of actual memory you have and the number of other Macintosh applications you wish to run at the same time. See the section "Setting the Size of Application Icons That Use Ivory".

An additional concern when drawing text or graphics into a bitmap is who does the drawing. When the bitmap is in host memory, the Macintosh host processor does the drawing, using QuickDraw. When the bitmap is in Ivory memory, the Ivory processor does the drawing. Which way is faster depends on the type of graphic operation being performed and the hardware configuration. In sketch mode (which is the default), the appearance of some graphic shapes varies depending on which processor does the drawing. For accurate drawing, you can disable sketch mode, but drawing into bitmaps in host memory (whether on-screen or off-screen) may then be slower. See the special form **graphics:with-scan-conversion-mode**.

These considerations mean the decision of whether to place an off-screen bitmap in Ivory memory or in host memory depends on the application and on the hardware configuration. The Genera window system decides as follows: Off-screen bitmaps for temporary use always go in host memory, unless not enough host memory is available even after swapping out other bitmaps. In this case they go in Ivory memory. Off-screen bitmaps for deexposed windows go in host memory or Ivory memory depending on how much host memory is available.

See the variable **mtb:*pixmap-correspondence-memory-threshold***. See the variable **mtb:*automatically-generate-pixmap-correspondences***.

Using Off-Screen Bitmaps in Your Application

Several interfaces are provided through which your application can use off-screen bitmaps. In embedded systems, all of these interfaces can store the off-screen bitmaps in host memory.

If you are presently doing things such as drawing to bit arrays and then **bitblt**ing to the screen, you should seriously consider using off-screen bitmaps in host memory, due to the substantial performance advantage in embedded systems such as MacIvory or the UX.

The available interfaces are:

tv:with-off-screen-drawing
tv:with-output-to-bitmap-stream
tv:with-output-to-bitmap
tv:allocate-bitmap-stream
tv:with-temporary-sheet-bit-raster

Using Off-Screen Bitmaps for Instantaneous Updates

This simple interface gives the appearance of an instantaneous update, rather than of each item appearing to be drawn separately. **tv:with-off-screen-drawing** works by copying the contents of a window to an off-screen bitmap, drawing into the bitmap, and copying back. In embedded systems, the off-screen bitmap always resides in host memory, provided sufficient host memory is available.

For instance,

```
(defun off-screen-strings ()
  (tv:with-off-screen-drawing (*terminal-io*)
    (dotimes (i 15)
      (write-string "12345678901234567890")
      (terpri))))

(defun off-screen-graphics ()
  (tv:with-off-screen-drawing (*terminal-io*)
    (graphics:with-room-for-graphics ()
      (graphics:draw-circle 100 100 50)
      (graphics:draw-triangle 20 0 120 0 70 75 :alu :flip))))
```

This involves some overhead in that you need to allocate and deallocate an off-screen bitmap, and copy from the window to the bitmap and back, each time the screen is visibly updated. In some cases the copy from the window to the bitmap is unnecessary, because the entire contents of the window will be redrawn. The **:complete-redisplay** option allows for optimization of this case. For example,

```
(defun off-screen-graphics-2 ()
  (tv:with-off-screen-drawing (*terminal-io*
                               :complete-redisplay t)
    (graphics:with-room-for-graphics ()
      (graphics:draw-circle 100 100 50)
      (graphics:draw-triangle 20 0 120 0 70 75 :alu :flip))
    (send *terminal-io* :refresh-margins)))
```

This is faster, but the previous content of the window is erased, where the previous example just scrolled it upwards.

Using Off-Screen Bitmap Streams

tv:with-output-to-bitmap-stream is a powerful interface that gives you an off-screen bitmap that supports all of the stream output and graphical operations of windows. The **:host-allowed** keyword controls whether the off-screen bitmap is stored in host memory or Ivory memory. There is no automatic copying from a window to the bitmap and back; you use the **:bitblt** and **:bitblt-to-sheet** messages to do this if and when you need it. Note that **:host-allowed t** is ineffective unless you also use **:for-stream** to specify a stream that leads to a screen; otherwise **tv:with-output-to-bitmap-stream** doesn't know which of potentially many hosts to use.

For example,

```
(defun off-screen-lines-1 (&aux (stream *terminal-io*))
  (fresh-line stream)
  (multiple-value-bind (cx cy)
      (send stream :visible-cursorpos-limits)
    (tv:with-output-to-bitmap-stream (bitmap-stream
                                      :for-stream stream
                                      :host-allowed t)
      (loop for x below 100 by 10 do
        (send bitmap-stream :clear-window)
        (loop for y downfrom (- 100 x) by 2 repeat 5 do
          (loop for x from x by 2 repeat 5 do
            (send bitmap-stream :draw-line x 0 0 y)))
        (send bitmap-stream :bitblt-to-sheet boole-xor 100 100 0 0
              stream (+ cx 150) (+ cy 150))))))
```

The previous example used a bitmap the full size of the window. We can improve it by creating a bitmap only as large as we need:

```
(defun off-screen-lines-2 (&aux (stream *terminal-io*))
  (fresh-line stream)
  (multiple-value-bind (cx cy)
      (send stream :visible-cursorpos-limits)
    (tv:with-output-to-bitmap-stream (bitmap-stream
                                      :for-stream stream
                                      :host-allowed t
```

```
                                        :width 100 :height 101)
      (loop for x below 100 by 10 do
        (send bitmap-stream :clear-window)
        (loop for y downfrom (- 100 x) by 2 repeat 5 do
          (loop for x from x by 2 repeat 5 do
            (send bitmap-stream :draw-line x 0 0 y)))
        (send bitmap-stream :bitblt-to-sheet boole-xor 100 100 0 0
              stream (+ cx 150) (+ cy 150))))))
```

If you specify values for **:width** and **:height** that are too small, the bitmap automatically expands so it is at least large enough to hold all the bits drawn. Of course the expansion takes time, so it is preferable to specify accurate values for **:width** and **:height**.

We can further improve the speed by using **:draw-multiple-lines** in place of **:draw-line**, to cut down on overhead. It speeds up this particular example by only 7 percent since the line-drawing speed of the Macintosh is the limiting factor in either case.

```
(defun off-screen-lines-2-m (&aux (stream *terminal-io*))
  (fresh-line stream)
  (multiple-value-bind (cx cy)
      (send stream :visible-cursorpos-limits)
    (tv:with-output-to-bitmap-stream (bitmap-stream
                                      :for-stream stream
                                      :host-allowed t
                                      :width 100 :height 101)

      (loop for x below 100 by 10 do
        (send bitmap-stream :clear-window)
        (send bitmap-stream :draw-multiple-lines
          (loop for y downfrom (- 100 x) by 2 repeat 5
                nconc
                  (loop for x from x by 2 repeat 5
                        collect x collect 0
                        collect 0 collect y)))
        (send bitmap-stream :bitblt-to-sheet boole-xor 100 100 0 0
              stream (+ cx 150) (+ cy 150))))))
```

tv:with-output-to-bitmap is similar to **tv:with-output-to-bitmap-stream** with the additional feature that it copies the off-screen bitmap into a Lisp array and returns it. Use this when you wish to capture the result of some output operations in an array of bits, rather than (or in addition to) displaying the result on the screen.

If you need an off-screen bitmap stream with a more permanent lifetime, you can use explicit allocation and deallocation. See the function **tv:allocate-bitmap-stream**. See the function **tv:deallocate-bitmap-stream**. For example,

```
(defun off-screen-lines-3 (&aux (stream *terminal-io*))
  (fresh-line stream)
  (multiple-value-bind (cx cy)
      (send stream :visible-cursorpos-limits)
    (let ((bitmap-stream (tv:allocate-bitmap-stream
                                    :for-stream stream
                                    :host-allowed t
                                    :width 100 :height 101)))
      (loop for x below 100 by 10 do
        (send bitmap-stream :clear-window)
        (loop for y downfrom (- 100 x) by 2 repeat 5 do
          (loop for x from x by 2 repeat 5 do
            (send bitmap-stream :draw-line x 0 0 y)))
        (send bitmap-stream :bitblt-to-sheet boole-xor 100 100 0 0
              stream (+ cx 150) (+ cy 150)))
      (tv:deallocate-bitmap-stream bitmap-stream))))
```

In a more realistic example, the calls to **tv:allocate-bitmap-stream** and **tv:deallocate-bitmap-stream** would be in two different functions; otherwise **tv:with-output-to-bitmap-stream** would work just as well.

The bitmap stream can become invalid if the window system is shut down and started again. It is best not to retain an off-screen bitmap stream permanently, but only for the duration of one operation. To minimize the overhead of allocating and deallocating a bitmap stream, you can allocate it when your application window is exposed and deallocate it when your application window is deexposed.

Using Off-Screen Bitmaps with Low-level Drawing Primitives

If you only need a bitmap, without the stream output and graphical operations of windows, either because you are using the low-level drawing primitives, or because you only do **bitblt**'s, you can use the lower-level allocation primitives. Note that if you are using **sys:%draw-***xxx*, you need to switch to **tv:sheet-draw-***xxx*, since the system must be given a sheet in order to find the screen that connects to the host. There is no significant overhead in this switch.

```
(defun off-screen-lines-4 (&aux (sheet *terminal-io*))
  (tv:with-temporary-sheet-bit-raster (bitmap sheet 100 100)
    (loop for x below 100 by 10 do
      (tv:sheet-force-access (sheet)            ;Make sure has a screen array
        (letf (((tv:sheet-screen-array sheet) bitmap))  ;Use this one temporarily
          (tv:sheet-draw-rectangle 100 100 0 0 boole-andc1 sheet)
          (loop for y downfrom (- 100 x) by 2 repeat 5 do
            (loop for x from x by 2 repeat 5 do
              (tv:sheet-draw-line x 0 0 y boole-ior nil sheet))))
        (tv:sheet-bitblt boole-xor 100 100 bitmap 0 0 nil 200 200 sheet)))))
```

If you need an off-screen bitmap with a more permanent lifetime, you can use explicit allocation and deallocation. Instead of using **tv:with-temporary-sheet-bit-**

raster, you can call **tv:%screen-allocate-sheet-temporary-bit-array** and **tv:%screen-deallocate-sheet-temporary-bit-array**. Note that all these programs work on the XL and the 3600 series as well.

For further information:

See the macro **tv:with-off-screen-drawing**.
See the function **tv:%screen-allocate-sheet-temporary-bit-array**.
See the function **tv:%screen-deallocate-sheet-temporary-bit-array**.
See the macro **tv:with-temporary-sheet-bit-raster**.

tv:allocate-bitmap-stream &key *:for-stream :host-allowed :width :height :bits-per-pixel :graphics-transform* *Function*

Allocates an off-screen bitmap stream. You can perform textual and graphic output operations on the stream returned. The results will not be visible, since the bitmap is off-screen. You can use the **:bitblt-to-sheet** message to make the result visible by copying from the off-screen bitmap to a window.

The bitmap stream can become invalid if the window system is shut down and started again. It is best not to retain an off-screen bitmap stream permanently, but only for the duration of one operation. To minimize the overhead of allocating and deallocating a bitmap stream, you can allocate it when your application window is exposed and deallocate it when your application window is deexposed.

:for-stream A stream that outputs to a related window. This provides defaults for the width, height, depth, and coordinate transformation; if **:host-allowed t** is specified, the host owning the window's screen will be used.

:host-allowed True if the off-screen bitmap should be stored in host memory if possible. The default is false, which always uses Ivory memory. **:host-allowed t** only works if **:for-stream** is specified.

:width The initial width of the bitmap. It expands if necessary.

:height The initial height of the bitmap. It expands if necessary.

:bits-per-pixel The depth of the bitmap.

:graphics-transform

A coordinate transformation. The default is the stream's transform if **:for-stream** is specified. Otherwise, the identity transform is the default.

See the message **:bitblt-to-sheet**.

See the function **tv:with-output-to-bitmap-stream**.

tv:deallocate-bitmap-stream *bitmap* *Function*

Deallocates an off-screen bitmap stream when you are no longer using it. See the function **tv:allocate-bitmap-stream**.

tv:bitmap-stream-copy-bitmap *stream* *Function*

Makes a copy of the bitmap or raster associated with a stream and returns five values:

- The bitmap
- Left top
- Right top
- Left bottom
- Right bottom

:bitblt-to-sheet *alu width height x y sheet sheet-x sheet-y* *Message*

Sends this message to an off-screen bitmap to copy its contents onto a window where it will be visible. This performs a **bitblt** from the *width* by *height* rectangle of the bitmap whose top-left corner is at *(x,y)* to the *width* by *height* rectangle of the window *sheet* whose top-left corner is at *(sheet-x,sheet-y)*.

See the function **bitblt**.

See the function **tv:allocate-bitmap-stream**.

See the function **tv:with-output-to-bitmap-stream**.

mtb:*automatically-generate-pixmap-correspondences* *Variable*

If *true* (the default), off-screen bitmaps that save the images of deexposed windows are placed in host memory if there is enough host memory. See the variable **mtb:*pixmap-correspondence-memory-threshold***.

If *false*, off-screen bitmaps that save the images of deexposed windows are always placed in Ivory memory.

You must set this variable before the window system starts up. Typically, you would set it before saving a world.

mtb:*pixmap-correspondence-memory-threshold* *Variable*

Determines how much host memory is considered enough to enable off-screen bitmaps that save the images of deexposed windows to be placed in host memory. The amount of available host memory at the time the window system starts up

must be at least **mtb:*pixmap-correspondence-memory-threshold*** times the size of the screen.

tv:with-output-to-bitmap *(*&optional *stream* &key *:for-stream :graphics-transform)* &body *body* *Function*

stream The stream to which to return the bitmap.

:for-stream
The stream for which the bitmap is intended.

:graphics-transform
An optional transform to be applied.

Returns a raster array and positions containing the image output by *body*.

```
(defun bitmap-example (&optional (stream *standard-output*))
  (graphics:with-room-for-graphics ()
    (graphics:draw-triangle 0 0 200 0 50 50
        :tile (tv:with-output-to-bitmap (bstream
                    :for-stream stream)
                (graphics:draw-circle 0 0 10
                    :gray-level .25 :stream bstream)
                (graphics:draw-regular-polygon 8 0 16 0 6
                      :gray-level .75
                    :stream bstream)))))
```

tv:with-output-to-bitmap-stream (*bitmap-stream* &rest *args* &key (*for-stream* **nil**) &allow-other-keys) &body *body* *Function*

bitmap-stream
A stream that is a raster array intended to hold the image generated by *body*.

args **:for-stream**, the stream for which the bitmap is intended, and, optionally, **:graphics-transform**, an optional transform to be applied.

Binds *bitmap-stream* to a specially allocated stream that accepts the graphic output during execution of *body*. At any time, the **:bitmap-and-edges** message to this stream returns the current image.

Batching Graphics Requests for MacIvory

If you are drawing a number of graphics to a MacIvory screen (or remote off-screen bitmap), you may benefit from batching them together into single larger requests to the Macintosh. This does not cut down the actual drawing time, but reduces the fixed communication overhead. This batching is only supported for lines and rectangles.

There are two new messages defined on windows (including those not attached to a MacIvory), **:draw-multiple-lines** and **:draw-multiple-rectangles**. Note that **:draw-multiple-lines** is different from **:draw-lines**, in that each line is specified by points, rather than joining to the previous line segment.

If you are using the higher level **graphics:draw-line** and **graphics:draw-rectangle**, you can still benefit from this batching by telling the system to buffer successive draw-line or draw-rectangle requests together. You will sacrifice a little bit of latency for better overall performance. To do this, **(setf (rpc::rpc-screen-buffered-graphics-enabled** <*screen*>**) t)**.

Direct Calls: a Linking Feature for Ivory-based Machines

The Ivory architecture provides Direct Calls, a fast mechanism for function calls that is mostly usable for benchmarking and application delivery.

In a normal Lisp call (an "indirect" call), the caller function has a pointer to the *function cell* containing the function to be called. When the call instruction is executed, it fetches the callee function from the function cell, and starts execution at the entry instruction of that function. The entry instruction sequence checks that the proper number of arguments was passed, initializes optional and keyword arguments, and then proceeds to execute the body of the called function.

The normal call is called "indirect" because it fetches the contents of a function cell (indirects through it) rather than addressing the callee function directly. Lisp implementations typically implement calls as indirect calls in order to efficiently support redefinition at runtime: When a function is redefined, all the Lisp system has to do is change the contents of the function cell, and all callers will immediately address the new definition.

In a direct call, the caller addresses the callee function directly, without going through a function cell. For Lisp systems that implement function calls using the direct method, redefinition must change *every* caller of a function to address the new definition. This is typically very slow.

Another optimization is possible when calls are implemented directly. Since relatively simple static analysis can determine how many arguments are being passed to a function, a direct call can often skip the preamble instructions that check for the proper number of arguments and initialize optional arguments.

Genera 8.0 provides a linker for Ivory-based machines that performs both of the above optimizations. Depending on the application, its use can result in substantial performance improvements. The linker is not fully integrated with Genera. If there are direct calls to a function, and there is an attempt to redefine it, an error is signaled. Proceed options allow you to unlink definitions to a function before redefining, or to proceed without unlinking.

To globally link all functions, use `(cli::link-to-functions t)`. To globally unlink them, use `(cli::unlink-to-functions t)`. If you need finer control of which existing functions should be linked or unlinked, refer to **cli::link-to-functions** and **cli::unlink-to-functions** for further information.

Regardless of whether any functions are linked or not, newly compiled or loaded functions are always unlinked.

Note: Because of architectural limitations, linking does not work on 3600-family machines, In order to get the additional performance benefit of linking, you must use an Ivory-based processor.

cli::link-to-functions *functions* &optional *link-noter verbose* *Function*

Links all calls to the functions specified by *functions*. *functions* is either a list of functions and/or function specs, or the symbol **t**, meaning all functions. This process takes up to twenty minutes, depending on your system configuration and the amount of software loaded.

cli::unlink-to-functions *functions* &optional *unlink-noter verbose* *Function*

Unlinks all calls to the functions specified by *functions*. *functions* is either a list of functions and/or function specs, or the symbol **t**, meaning all functions. This process takes about five minutes, depending on your system configuration and the amount of software loaded.

Converting User Interfaces for MacIvory

As a matter of principle, we recommend that you convert your programs in stages. If you wish, you can also convert your code by making all the source changes at once and then getting your program to run. However, this is generally a less efficient and more tedious process.

You should be generally familiar with the user-level operation of the MacIvory and the high-level user-oriented description of how the MacIvory user interface works.

- First, make sure that your program runs under Genera 7.2 on the 3600-series. Genera 7.2 is almost completely compatible with earlier releases. If you discover a large number of incompatibilities in converting to Genera 7.2, you are probably using many undocumented internal interfaces. If nothing else, this should alert you to parts of your program that may run into problems on the MacIvory.

- Make the changes necessary to get your program to run on the Ivory architecture. The Ivory is almost entirely source level compatible, so this should not be a major undertaking.

- Genera 8.0 Ivory includes a number of changes to the scheduler and window system locking originally made in Genera 7.9 Ivory of which you should be aware. For further information, see the section "Converting to the New Scheduler". Get your program to run on the MacIvory with minimal changes. There is only one major thing that is completely different on the MacIvory. There is no virtual memory array on the Lisp side that maps to the screen array of an exposed window. All drawing is done on the Macintosh side. If you are using

tv:sheet-screen-array or the **tv:screen-array** instance variable of a sheet, you need to change to go through the screen. This conversion is straightforward. Instead of the very low level **sys:%draw-xxx** functions, you should use the **tv:sheet-draw-xxx** functions. For instance, these forms:

```
(bitblt tv:alu-seta 100 100 raster 0 0 (tv:sheet-screen-array sheet) 100 100)
(bitblt tv:alu-seta 100 100 (tv:sheet-screen-array sheet) 0 0 raster 100 100)
```

become

```
(tv:sheet-bitblt tv:alu-seta 100 100 raster 0 0 nil 100 100 sheet)
(tv:sheet-bitblt tv:alu-seta 100 100 nil 0 0 raster 100 100 sheet)
```

and this form:

```
(tv:%draw-char-internal #o101 fonts:43vxms 10 10 tv:alu-xor tv:screen-array)
```

becomes

```
 (tv:sheet-draw-glyph #o101 fonts:43vxms 10 10 tv:alu-xor)
```

- The other incompatibility you may experience from the very start is the change in the screen size. If you think this will be a major problem, one good way to get started is to use a larger Macintosh monitor for the initial conversion. There are a variety available that approximate the size and pixel density of the 3600-series screens.

MacIvory differs from previous systems in that screen drawing is not done by the same processor that executes Lisp programs. Ivory executes Lisp, but the Macintosh does the screen drawing. When you execute a graphics or text output function or message, on 3600-series systems it does not return until the screen drawing is complete, but on MacIvory it queues a request for the Macintosh to do the screen drawing and returns immediately. If you have a program that needs to synchronize itself with the screen drawing, use the **finish-output** function (also available as the **:finish** message).

On 3600-series machines, **finish-output** on a window takes no appreciable time, but on MacIvory it waits until previously queued screen output has been completed. An example of when you would need to do this is a program that draws continuously while a mouse or keyboard button is held down, and is supposed to stop as soon as the button is released. Without **finish-output**, the Ivory may stop as soon as the button is released, but the already-queued output will continue to be drawn for a noticeable amount of time.

Bitblt operations where the source and destination are the same bitmap in host memory (which could be a window's image or an off-screen bitmap), and the source and destination subrectangles overlap, are not compatible with the 3600 and XL400 native window system. The embedded window system always chooses the direction of transfer so as to shift the contents of the bitmap. The native window system requires the caller to use the sign of the width and height arguments to

specify the direction of transfer, either shifting the contents of the bitmap or smearing it. In effect, you cannot use **:bitblt-within-sheet** for smearing effects on the embedded window system.

Once your program has been ported to MacIvory, you should decide how to best make use of the system.

- If your output uses the **graphics:draw-***xxx* drawing functions it will use the QuickDraw drawing primitives directly. This can give a performance improvement of as much as 100 times. There are some incompatibilities, of course:

  ° Stipples will always be 8x8. If yours do not have that phase, they may not align properly.

  ° Unfilled shapes are stroked down and to the right of the mathematical shape with a rectangular pen, rather than half on either side.

  To make output exactly compatible, use the **graphics:with-scan-conversion-mode** special form. (See the special form **graphics:with-scan-conversion-mode** for further information.)

- Keep in mind that coordinate system transformation is still performed on the Lisp side and that some shapes, such as tilted ellipses and dashed lines, are not supported by QuickDraw and therefore are still handled the slower, compatible way. As a general rule, however, if your graphics output already works compatibly to the LGP2/LGP3, you will probably do fine using QuickDraw also.

- If your output is using **:draw-line** or **:draw-triangle** and requires the ability to draw endpoints or abut objects seamlessly. Then you must use **tv:screen-genera-graphics-compatibility** of the screen to which you are drawing to locally enable compatibility for these operations.

- If you are drawing to bit arrays and then copying them onto the screen to keep down the flicker time, you may wish to allocate these arrays on the Macintosh side and have the drawing done remotely. Then, the copying can also be done entirely on the Macintosh.

If you are not concerned with compatibility among platforms, you may wish to convert your program to have a Macintosh-style user interface. A version of **dw:define-program-framework** is provided which provides high-level access to the major capabilities. For further information, see the special form **dw:define-remote-program-framework**. If you have special peripherals as part of your interface, such as a tablet, or wish to have special output, such as sound, you may need to make use of the MacIvory RPC facilities which access the Macintosh toolkit. These are relatively easy to use, given a basic knowledge of the Macintosh. Lisp-level support is given to remove most of the tedious host programming that would be necessary to communicate between the processors.

You may also wish to add some specific high-level request communications to tailor the performance of your application to the co-processor environment. Many well-defined operations can be moved onto the host. For instance, if you require very responsive rubber-band line style mouse tracking, you can define a procedure in the Macintosh mouse handler to do this and pass the results back the Ivory. You may wish to consult your Symbolics contact for help in meeting your specific needs.

Example of Converting an Application for MacIvory

This section describes how an existing application program — Document Examiner — was converted to work in Genera Ivory on the MacIvory platform. The goal was to present essentially the same Document Examiner on both the MacIvory and Symbolics 3600-series computers.

In presenting this example we hope to illustrate some common problems in preparing the user interface for Symbolics applications on MacIvory and their solutions. Some general steps are:

- Recompiling the system for Genera Ivory.
- Identifying any problems in existing interface design.
- Redesigning the user interface, if necessary. Depending on the purpose of your application, you may need to rework elements to fit a smaller screen size, or to resemble a Macintosh style interface (described in the Macintosh User Interface Guidelines).
- Implementing parallel user interfaces.
- Refining the new implementation.

Recompiling

The first step is to recompile the system in a Genera Ivory world running on a MacIvory. (You can use the Compile System command or the **compile-system** form.) The recompilation occurred without any problems, producing a set of IBIN files. This application was not affected by any of the Genera Ivory changes.

Finding Problems with the Scaled-down Interface

At this point, Document Examiner works on the MacIvory. When we try it on both a large display and a small display the scaling of the user interface from the usual Symbolics display to the large display on the Macintosh is acceptable. However, there are several problems with the scaled-down user interface on the small display.

The first problem occurs when `SELECT D` on a small screen results in an error. The cause of this error is that the size of the Viewer pane has been specified with

a fixed number of pixels, which exceeds the width of the small screen. When we change the constraints from a fixed number of pixels to proportional constraints, the immediate problem is solved.

Document Examiner makes full use of a large screen. Each pane is important and necessary. When scaled, the two biggest problems are:

- The formatted text in the Viewer pane depends on larger margins.
- The appearance of the screen is excessively crowded.

The Viewer shows the online documentation. The text was formatted expressly for a large screen, and when it is scaled down for the small screen, the text disappears beyond the right edge of the pane. This means that to read any given line, you would have to scroll the window toward the right, and then scroll back to the left to read the beginning of the next line.

One solution might be to change the formatter code to format differently, according to the size of the display. Another possible solution is to redesign the user interface of Document Examiner for the small screen. Since it was a goal to have Document Examiner work for both the large and small screens, either solution would require two parallel versions of some code, each version targeted for a particular size of display. (Either the formatter would work differently, based on the display, or the user interface would work differently.)

We decided that redesigning the screen was preferable to changing the formatter. Using this approach we could solve both problems. Since the primary purpose of Document Examiner is to present documentation in a readable way, we wanted to grant more of the screen to the text being displayed. Along the way, we would have to reduce the amount of screen granted to other panes.

To pursue this solution, we do the following:

- Design a user interface targeted for the small screen
- Find a way to choose the user interface at run-time, based on the size of the screen

Designing the User Interface for the Small Screen

Figure ! shows how Document Examiner appears on the large screen.

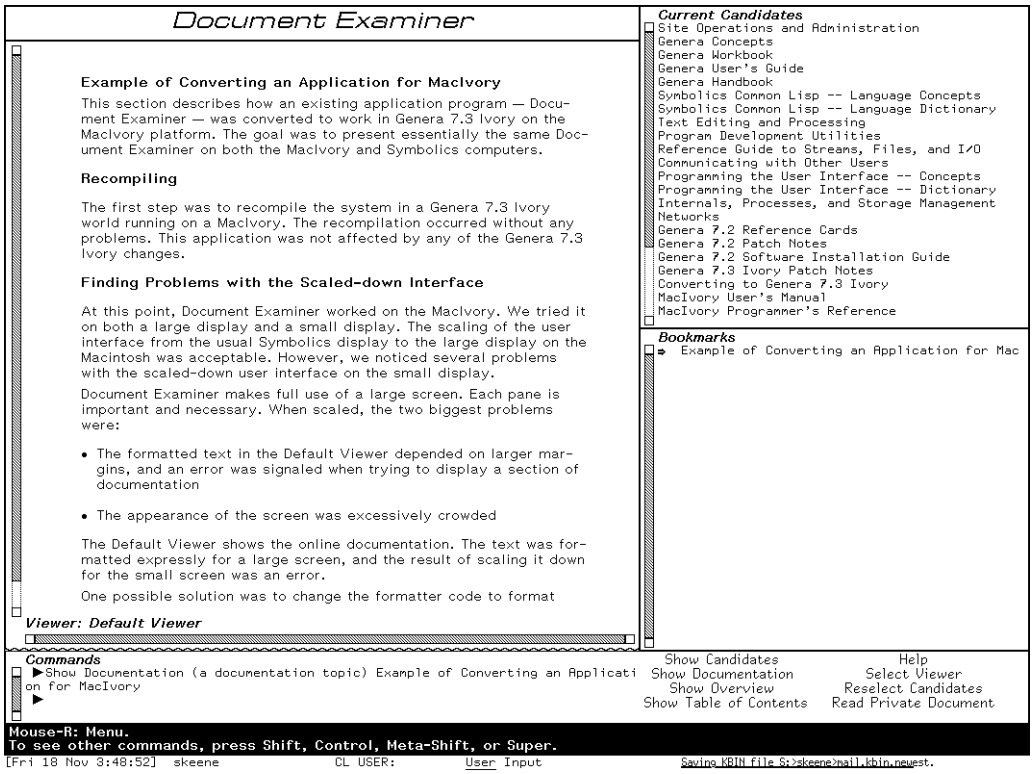

Figure 95. Large-screen Appearance of Document Examiner

We decided to use the entire width of the small screen for the Default Viewer. We placed the Current Candidates and Bookmarks panes below the Default Viewer, and the Command interaction pane below them. We reduced the size of these three panes in order to compensate for increasing the size of the Default Viewer pane.

The Command Menu pane was the only remaining pane. Instead of dedicating space on the screen for it, we decided to make Document Examiner commands available via Macintosh-style pull-down menus. We created two new pull-down menus, called DocEx and Show, and we placed Document Examiner commands in these menus.

Figure ! shows how Document Examiner appears on the small screen. Note that the two figures have the same scale factor, so you can get an idea of the difference in the sizes of the two screens.

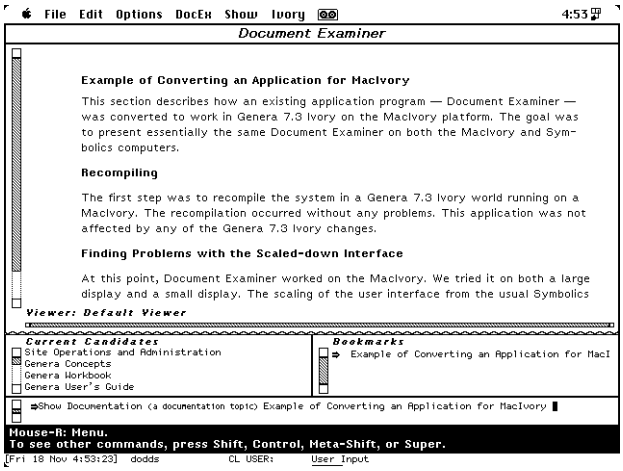

Figure 96. Small-screen Appearance of Document Examiner

The result is two different displays and modes of interaction, one for the large screen (which was unchanged from its existing state), and one for the small screen. Even though the mechanics of giving Document Examiner commands are slightly different, the general look-and-feel of the two interfaces are so similar that we believe users could easily switch between them.

Implementing the Parallel User Interfaces

We needed to provide two **dw:define-program-framework** forms, one for each screen. The form for the large screen already existed.

In both forms, it was important to make the **:selectable** option be **nil**, the **:system-menu** option **nil**, and not to provide the **:select-key** option. As shown later, we will use a different mechanism for choosing the appropriate user interface at run-time.

The primary difference in the two forms is in the **:configuration** option, highlighted here:

```
(dw:define-program-framework doc-ex
  :pretty-name "Standard Document Examiner"
  :system-menu nil
  :selectable nil
  other-options...
  :configurations
  '((dw::main
      (:layout (dw::main :column top-part bottom-part)
       (top-part :row title&viewer-pane candidates&bookmarks)
       (bottom-part :row command-pane menu-pane)
       (title&viewer-pane :column title-pane viewer-pane)
       (candidates&bookmarks :column candidate-pane bookmark-pane))
      (:sizes
         (dw::main (bottom-part 4 :lines command-pane)
                   :then (top-part :even))
         (bottom-part (command-pane 660.)
                      :then (menu-pane :even))
         (top-part (title&viewer-pane 660.)
                   :then (candidates&bookmarks :even))
         (title&viewer-pane (title-pane 1 :lines)
                            :then (viewer-pane :even))
         (candidates&bookmarks (candidate-pane 0.5)
                               :then (bookmark-pane :even))))
       ))

(dw:define-program-framework small-doc-ex
  :pretty-name "Small Document Examiner"
  :system-menu nil
  :selectable nil
  other-options...
  :configurations
  '((dw::main
      (:layout (dw::main :column title&menu viewer-pane
                         candidates&bookmarks command-pane)
       (title&menu :row title-pane #+Ignore menu-pane)
       (candidates&bookmarks :row candidate-pane bookmark-pane))
      (:sizes
         (dw::main (command-pane 2 :lines) (title&menu 1 :lines title-pane)
                   (candidates&bookmarks 4 :lines candidate-pane)
                   :then (viewer-pane :even))
         (title&menu (title-pane :even))
         (candidates&bookmarks (candidate-pane 0.5)
                               :then (bookmark-pane :even)))
      )))
```

Causing the Correct User Interface to be Selected at Run-time

In the example below, we define the function **choose-doc-ex-for-screen** to perform a run-time query to determine the size of the screen. The result of this query selects the appropriate Document Examiner.

The **choose-doc-ex-for-screen** function is "hooked into" the activity selection mechanism by using **tv:add-dispatching-select-key**, a function documented for general use. The two function calls that follow do some additional cleanup work. These functions are not documented other than in this example.

However, for now, if you need to choose the application based on run-time conditions, you can follow this example.

```
(defun choose-doc-ex-for-screen (screen)
  (let ((screen-type
          (if (> (send (or screen tv:main-screen) :inside-width)
                 950.)
              :big-screen
              :small-screen)))
    (ecase screen-type
      (:big-screen 'doc-ex)
      (:small-screen 'small-doc-ex))))

(tv:add-dispatching-select-key
  #\D 'choose-doc-ex "Document Examiner" 'choose-doc-ex-for-screen t)

;;; The following is necessary to put the "Document Examiner" entry
;;; in the system menu, as D-P-F would do.
(tv:add-activity-to-system-menu-programs-column "Document Examiner")
(tv:add-to-system-menu-create-menu
  "Document Examiner"
  '(program-frame :program choose-doc-ex)
  "A Document Examiner program frame.")
```

Conditionalizing the Prompt

This final piece of code fine-tunes the application. The problem it addresses is that the usual Document Examiner prompt was too large for the small screen. This is because the prompt uses a device font, and device fonts are always the same size. They do not use the normal character-style mechanism, and are not automatically scaled down for the smaller screen. This code remedies the problem and selects the Document Examiner prompt based on the program running in the window.

```
(defun type-of-docex-frame (window)
  (dw::program-name (send (send window :alias-for-selected-windows)
                          :program)))
```

```
(defun default-dex-prompt (stream keyword)
  (ignore keyword)
  (with-character-style ('(:fix :roman :normal) stream)
    (ecase (type-of-docex-frame stream)
      (small-doc-ex
        (send stream :tyo #\space)
        (send stream :tyo #\arrow:right-open-arrow))
      (doc-ex
        (send stream :tyo #\space)
        (send stream :tyo #\arrow:right-triangle)))))
```

MacIvory User Interface Examples

Etest Example

etest.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;;> EXAMPLES-MESSAGE
;;;>
;;;>*****************************************************************************************
;;;>
;;;>       Symbolics hereby grants permission to customers to incorporate
;;;>       the examples in this file in any work belonging to customers.
;;;>
;;;>*****************************************************************************************

(dw:define-remote-program-framework etest
  :selectable (:mac)
  :menu-level-order `(,macintosh-internals::*standard-remote-viewer-file-menu*
                      ,macintosh-internals::*standard-edit-menu*
                      "ETest")
  :inherit-from (;; Enable Lisp event handling
                 mtb::lisp-handles-event-uims
                  macintosh-internals::remote-viewer-commands)
  :command-table (:kbd-accelerator-p t :inherit-from '("remote-viewer-commands"
                                                       ;; Handle some events
                                                       "Mac Window Control Commands"))
  :state-variables ((splash-window))
  )
```

```
;; This scrawls a big black circle, showing that scroll bar tracking
;; with Lisp callbacks works.
(define-etest-command (com-test-1
                        :menu-accelerator "Test-1"
                        :menu-level (:top-level (:mac "ETest"))
                        :keyboard-accelerator #\s-S)
    ()
  (let ((w (dw::remote-program-open-viewer self :picture-p nil
                                           :left 40 :top 40 :width 600 :height 600
                                           :title "This is a test"
                                           :displayer 'display-etest-startup-window)))
    (send w :expose)                            ;only this creates the Macintosh side
    (multiple-value-bind (scroll-bar-h scroll-bar-v)
        (macintosh-internals::mac-rpc-window-scroll-bars w)
      (mtb:_setctlmax scroll-bar-h 600)
      (mtb:_setctlmax scroll-bar-v 600))
    (mtb::force-update-window w)                ;---this shouldn't be needed
    ))

(defun display-etest-startup-window (w mac-window)
  (ignore mac-window)
  (graphics:draw-circle 200 200 100 :stream w))

(define-etest-command (com-splash :menu-accelerator "Splash"
                                  :menu-level (:top-level (:mac "ETest"))
                                  :keyboard-accelerator #\s-F)
    ()
   (let ((w (if (member splash-window
                        (macintosh-internals::mac-rpc-program-windows self))
                splash-window
                (setq splash-window
                      (dw::remote-program-open-viewer self :picture-p nil
                                   :left 5 :top 30 :width 470 :height 370
                                   ;; This title isn't really called
                                   ;; for by the graphic design...
                                   :title "BearCal Splash"
                                   :displayer (etest-splash-displayer))))))
      (send w :expose)                          ;only this creates the Macintosh side
      (mtb::force-update-window w)              ;---this shouldn't be needed
      ))
```

```
(defun etest-splash-displayer ()
  (let* ((inner-pict-handle ())
         (inner-pict-rect ())
         (displayer
           (sys:named-lambda splash-displayer (ignore ignore)
             (when (null inner-pict-handle)
               ;; The closure state needs some initialization
               (setq inner-pict-rect
                     (mtb:make-rect :left 5 :top 30 :right 475 :bottom 400))
               ;; Pre-scribble into a pict so we won't need file I/O at redisplay time.
               (setq inner-pict-handle (mtb:_openpicture inner-pict-rect))
               (with-open-file (in "sys:embedding;macivory;toolkit;examples;Splash.pict"
                                   :element-type '(unsigned-byte 8))
                 ;; Open code mtb::with-pict-from-file because it's on the sys host
                 ;; instead of on the local Mac.
                 (let ((len (- (send in :length) 512)))
                   ;; The first 512 bytes of a pict file are useless.  The rest is the pict.
                   (send in :set-pointer 512)
                   (stack-let ((pic (make-array len :element-type '(unsigned-byte 8))))
                     ;; Read it from sys host into Lisp
                     (send in :string-in "reading pict data" pic)
                     ;; Ship it over to the Mac.
                     (mtb:with-mac-temp (pict :handle T :initial-contents pic)
                       (let ((pict-rect (mtb:octet-ref-remote-handle
                                          (mtb:picture-picframe pict))))
                         (mtb:_offsetrect pict-rect -20 -20)
                         (mtb:_drawpicture pict pict-rect))))))
               (mtb:_textsize 18)
               (mtb:_textface (mtb:cconstant bold))
               (mtb:_moveto 100 20)
               ;; The octal 245 here is the Macintosh center-bullet char
               (mtb:_drawstring "Welcome to BearCal Airlines")
               (mtb:_moveto 115 330)
               ;; The 322 and 323 are Macintosh sex-differentiated quotes
               (mtb:_drawstring "You'll have reservations...")
               (mtb:_closepicture))
             ;; This is what really does the display
             (mtb:_drawpicture inner-pict-handle inner-pict-rect))))
    displayer))
```

```
(defvar *splash-resource-file-pathname* #p"host:dsk:rlb:credit-card-picts.rsrc")
;;; Since PICT dialog items are identified by PICT resource numbers,
;;; they must live in a resource file.  We distribute a little rsrc file with the
;;; picts for this example, but it's unlikely your Macintosh can read them from the sys host.
;;; So, to run this example, you should do:
;;; (mtb:copy-mac-image "sys:embedding;macivory;toolkit;examples;credit-card-picts.rsrc"
;;;                     "host:dsk:your-folder:credit-card-picts.rsrc")
;;; to move the picts to your Macintosh's file system, and edit the defvar above,
;;; changing "rlb" to reflect its new home.

(define-etest-command (com-dialog
                       :menu-accelerator "Dialog"
                       :menu-level (:top-level (:mac "ETest"))
                       :keyboard-accelerator #\s-D)
    ()
   (mtb:with-resfiles ((ignore *splash-resource-file-pathname*))
     (let ((values (mtb:do-modal-dialog self (etest-dialog-items))))
       ;;---of course this is all bogus
       (let ((results (with-output-to-string (s)
                        (loop for (k v) on values by #'cddr do
                          (when (and v (not (equal v "")))
                            (format s "  ~a: ~a~%" k v))))))
         (unless (zerop (length results))
           ;; This is just a way to pop up something in lieu of updating
           ;; some Statice database
           (macintosh-internals::display-dialog-help
             self
             (format nil "Confirm:~%~a" results)
             (mtb::coerce-to-rect '(30 50 30 50))
             500 400))))))

(defun etest-dialog-items ()
  (let ((m (mtb::make-dialog-item-maker)))
    (mtb::set-dialog-face m
                          :bounds '(30 50 482 335)
                          :title "BearCal Reservation System"
                          :proc-id (mtb:cconstant rDocProc))

    (mtb::add-several-dialog-items
      m
      '((:button (380 10 440 30) "OK"
                 :check T :cluster okay-cluster :query-id okay-button)
        (:button (380 40 440 60) "Cancel"  :query-id cancel-button)
```

```
    (:text (10 10 110 26) "Passenger:")
    (:edit (115 10 360 26) :query-id passenger :required T
           :oversee-cluster okay-cluster)

    (:text (10 35 110 51) "Destination:")
    (:edit (115 35 360 51) :query-id destination :required T
           ;; :format1 "A" :format2 "["
           )

    (:text (10 60 110 76) "Depart:")
    (:edit (115 60 185 76) :query-id depart :required T)

    (:text (205 60 285 76) "Return:")
    (:edit (290 60 360 76) :query-id return)

    (:line (12 85 440 85))))

(mtb::add-several-dialog-items
  m
  '((:text (10 95 110 111) "Address:")
    (:edit (115 95 440 111) :query-id address)

    (:text (10 120 110 136) "City:")
    (:edit (115 120 330 136) :query-id city)

    (:text (345 120 400 136) "State:")
    (:edit (405 120 440 136) :query-id state)

    (:text (10 145 110 161) "Zip:")
    (:edit (115 145 220 161) :query-id zip)

    (:text (235 145 295 161) "Phone:")
    (:edit (300 145 440 161) :query-id phone)

    (:line (12 170 440 170))
    (:text (370 180 440 196) "Payment:")
    (:cluster payment-cluster
     (:pict (375 200 434 234) 100
            :query-id supercard)
     (:pict (375 240 434 274) 101
            :query-id creditcard))

    (:line (360 175 360 275))))
```

```
(mtb::add-several-dialog-items
  m
  '((:text (10 180 110 196) "Incidentals:")
    (:cluster incidentals-cluster
     (:check (15 200 110 216) "Smoking"  :report T
             :query-id incidental-smoking)
     (:check (15 220 110 236) "Dinner"
             :query-id incidentals-dinner)
     (:check (15 240 110 256) "Movie"
             :query-id incidentals-movie)
     (:check (15 260 110 276) "Rental Car"  :report T
             :query-id incidental-rental-car))))

(mtb::add-several-dialog-items
  m
  '((:text (130 180 230 196) "Class")
    (:cluster class-cluster
     (:radio (135 200 230 216) "First"
             :query-id first-class)
     (:radio (135 220 230 236) "Coach"
             :query-id coach-class)
     (:radio (135 240 230 256) "Nook"
             :query-id nook-class))))

(mtb::add-several-dialog-items
  m
  '((:text (250 180 350 196) "Seating")
    (:cluster seating-cluster
     (:radio (255 200 350 216) "Window"
             :query-id seating-window)
     (:radio (255 220 350 236) "Center"
             :query-id seating-center)
     (:radio (255 240 350 256) "Aisle"
             :query-id seating-aisle))))
m))
```

Note: credit-card-picts.rsrc and splash.pict are binary data files used by the ETest commands. splash.pict is taken from the sys host, so you can try it out without further work. To use the credit-card-picts.rsrc demo, you must copy it onto your Macintosh before running the Dialog demo, and then edit a **defvar** in etest to say where the file has been copied.

Pic-show Example

pic-show.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: (PIC-SHOW (MTB MACINTOSH-INTERNALS RPC SCL)); Base:
0 -*-
;;;> EXAMPLES-MESSAGE
;;;>
;;;>*****************************************************************************************
;;;>
;;;>       Symbolics hereby grants permission to customers to incorporate
;;;>       the examples in this file in any work belonging to customers.
;;;>
;;;>*****************************************************************************************

(dw:define-remote-program-framework pic-show
  :selectable (:mac)
  :menu-level-order `(,macintosh-internals::*standard-remote-viewer-file-menu*
                      ,macintosh-internals::*standard-edit-menu*
                      "PicShow")
  :inherit-from (;; Enable Lisp handling of events
                 mtb::lisp-handles-event-uims
                  macintosh-internals::remote-viewer-commands)
  :command-table (:kbd-accelerator-p t :inherit-from '("remote-viewer-commands"
                                                       ;; Handle some events
                                                       "Mac Window Control Commands"))
  :state-variables ((window-pictures))
  )

;;;=======================================
;;; Register file types and ways to read pictures from them
;;;=======================================

(defvar *picture-readers* ())

(defun file-types-readable-as-pictures ()
  (mapcar #'first *picture-readers*))

(defun note-picture-reader (file-type function)
  (pushnew (list file-type function) *picture-readers*
           :key #'first :test #'string=))

(defun find-picture-reader (file-type)
  (second (assoc file-type *picture-readers* :test #'string=)))
```

```
(defmacro define-picture-reader (function-name file-type arglist &body body)
  `(progn
     (defun ,function-name ,arglist
       (declare (sys:function-parent ,function-name define-picture-reader))
       ,@body)
     (note-picture-reader ',file-type ',function-name)))

;;;=======================================
;;; Temporarily bind clipping region
;;;=======================================

(defmacro with-clip-rect ((rect) &body body)
  `(with-clip-rect-1 ,rect (sys:named-lambda with-clip-rect () ,@body)))

(defun with-clip-rect-1 (rect continuation)
  (declare (sys:downward-funarg continuation))
  (multiple-value-bind (left top right bottom)
      (dw:box-edges (mtb::coerce-to-box rect))
    (with-temps ((old-clip _newrgn _disposergn)
                 (new-clip _newrgn _disposergn))
      (_getclip old-clip)
      (_setrectrgn new-clip left top right bottom)
      (mtb::unwind-protect-try
        (progn
          (_setclip new-clip)
          (funcall continuation))
        (_setclip old-clip)))))


;;;=======================================
;;; Here's the main command
;;;=======================================
```

```
(define-pic-show-command (com-show-file :menu-accelerator "Open"
                                        :menu-level (:top-level (:mac :file))
                                        :keyboard-accelerator #\s-O)
    ()
   ;; It would be nice to change the mouse cursor to a wristwatch while preparing to show.
   (multiple-value-bind (name pict)
       ;; Those types we know how to read and show. When there are more than
       ;; four types this will have to use a file type filter instead of type list.
       ;; See Inside Macintosh p. I-524.
       (let ((types (file-types-readable-as-pictures)))
         (multiple-value-bind (vrefnum name type)
             (ez-sfgetfile :x-and-y '(100 100) :types types)
           (when vrefnum                        ;NIL if Cancel
             (with-open-refnum (refnum (fs:parse-pathname name "host")
                                       :vrefnum vrefnum)
               (values name (funcall (find-picture-reader type) refnum))))))
     (when name                                         ;NIL if Cancel
       (multiple-value-bind (rpc-window mac-window)
           ;; Get a place to display this picture
           (get-the-right-window self pict name)
         (display-picture-window rpc-window mac-window)))))

;;;=======================================
;;; Getting the pictures displayed
;;;=======================================

;; A collection of things replicated for each file/picture/window/etc
(defflavor window-picture (rpc-window mac-window scroll-bars pict zoom pict-rect display-rect)
           ()
  :initable-instance-variables
  :readable-instance-variables)
```

```
(defmethod (get-the-right-window pic-show) (pict name)
  (let ((pict-rect (octet-ref-remote-handle (picture-picframe pict))))
    (multiple-value-bind (pleft ptop pright pbottom)
        (values (rect-left pict-rect 0)
                (rect-top pict-rect 0)
                (rect-right pict-rect 0)
                (rect-bottom pict-rect 0))
      (let ((width (- pright pleft))
            (height (- pbottom ptop)))
        (multiple-value-bind (max-right max-bottom)
            ;;---this should be based on something like screenBits, but that isn't
            ;;readily accessible here.  So pick something wrong, at least for large screens,
            ;;but which should at least leave the size box on the screen.
            (values 600 400)
          (let* ((dy 20)
                 (dx 4)
                 (number-of-windows (length window-pictures))
                 ;; Offset the windows a little each time.
                 ;; This doesn't really work when some windows have been closed.
                 (left (+ 15 (* dx number-of-windows)))
                 (top (+ 40 (* dy number-of-windows)))
                 (right (min max-right (+ left width)))
                 (bottom (min max-bottom (+ top height))))
            (let ((rpc-window
                    ;; This is what with-output-to-viewer does
                    (dw::remote-program-open-viewer
                      self :picture-p nil
                      :left left :top top :right right :bottom bottom
                      :title name
                      :displayer 'display-picture-window)))
              (send rpc-window :expose)         ;force creation on Macintosh side
              (let ((mac-window (macintosh-internals::mac-rpc-window-mac-window rpc-window)))
                (multiple-value-bind (scroll-bar-h scroll-bar-v)
                    (macintosh-internals::mac-rpc-window-scroll-bars rpc-window)
                  ;; The scroll bars need to know how big a range they're scrolling
                  ;; This scroll bar stuff certainly deserves to have been done automatically
                  ;; by dw::remote-program-open-viewer
                  (mtb:_setctlmax scroll-bar-h width)
                  (mtb:_setctlmax scroll-bar-v height)
                  ;; Stash away our data structures.
                  ;; --- This little application has no code to
                  ;;  realize when a window is closed.
                  (push (make-instance
                          'window-picture
                          :rpc-window rpc-window
                          :mac-window mac-window
                          :scroll-bars (list scroll-bar-h scroll-bar-v)
```

```
                          :pict pict
                          :zoom 1
                          :pict-rect pict-rect
                          :display-rect (make-rect :left 0 :top 0
                                                   :right width :bottom height))
                        window-pictures)
                  (values rpc-window mac-window))))))))))

;; This is the redisplayer.
(defun display-picture-window (rpc-window mac-window)
  (ignore mac-window)
  ;; forward this request to the controlling remote program.
  (display-picture-window-1 (macintosh-internals::mac-rpc-window-program rpc-window)
                            rpc-window))

(defmethod (display-picture-window-1 pic-show) (rpc-window)
  (let ((window-picture (find rpc-window window-pictures :key #'window-picture-rpc-window)))
    (when window-picture
      ;; forward it to the application data structure.
      (display-picture-window-2 window-picture))))

(defmethod (display-picture-window-2 window-picture) ()
  (with-qd-port (mac-window)
    ;; Draw the pict.  Conceivably this could be accelerated in some cases by
    ;; knowing something of the port's cliprgn.
    (_drawpicture pict display-rect)))

;;;=======================================
;;; Ways to read files.
;;;=======================================

;; Pict files written by MacDraw and many others
(define-picture-reader get-the-pict "PICT" (refnum)
  (with-pict-from-file (pict pict-rect :refnum refnum :temporary nil)
    (ignore pict-rect)
    pict))
```

```
;; MacPaint files are written by many paint programs.
;; The first 512 bytes are skipped.  Then there are 720 rows
;; each 72 bytes wide (where each row needs UnpackBits).
;; Fixed size, fixed resolution.
(define-picture-reader get-the-pntg "PNTG" (refnum)
  (let ((len (- (_geteof refnum) 512)))
    ;; We need a place to keep the file data to UnpackBits from
    (mtb:with-mac-temp (pntg :handle t :size len)
      (_setfpos refnum (cconstant fsFromStart) 512)
      (let ((err (with-handle-locked (pntg)
                   ;; Read the file contents, but only on the Macintosh side.
                   (_fsread-remote refnum (_ptrfromhandle pntg) len))))
        (unless (zerop err)
          ;; fsread-remote doesn't itself signal errors.
          (signal-mac-os-error err)))
      ;; and a place to keep the unpacked bits.
      (mtb:with-mac-temp (the-bits :handle T :size (* 720 72))
        (with-handle-locked (the-bits)
          (with-handle-locked (pntg)
            (_unpackbitsbyrows (_ptrfromhandle pntg)
                               (_ptrfromhandle the-bits)
                               72 720))
          (with-rect (r 0 0 (* 72 8) 720)
            (stack-let ((bitmap (make-bitmap
                                 ;; the-bits must be locked while this is used as Ptr
                                 :baseaddr (_ptrfromhandle the-bits)
                                 :rowbytes 72
                                 :bounds r)))
              (let* ((the-port (_getport))
                     (portrect (octet-ref-remote-ptr (grafport-portrect the-port)))
                     (portbits (octet-ref-remote-ptr (grafport-portbits the-port))))
                ;; Turn the bitMap into a Pict by drawing it.
                (let ((pict (_openpicture r)))


                  (with-clip-rect (r)
                    (_copybits bitmap portbits portrect portrect
                               (cconstant srcCopy) 0))
                  (_closepicture)
                  pict)))))))))
```

```
;; IDMP is a very simple file format for grayscale images.
;; 2 bytes, bits-per-pixel
;; 2 bytes, width
;; 2 bytes, height
;; rest of file, the data.
(define-picture-reader get-the-idmp "IDMP" (refnum)
  (let (bits-per-pixel width height)
    (mtb:with-mac-temp (data :handle T :size 0)
      (stack-let ((front (make-array 6 :element-type '(unsigned-byte 8)
                                       :fill-pointer 0)))
        (_fsread refnum 6 front)
        (setq bits-per-pixel (byte-swapped-8-aref-16 front 0)
              width (byte-swapped-8-aref-16 front 2)
              height (byte-swapped-8-aref-16 front 4)))
      (let ((len (/ (* width height bits-per-pixel) 8)))
        (_sethandlesize data len)
        (let ((err (with-handle-locked (data)
                     (_fsread-remote refnum (_ptrfromhandle data) len))))
          (unless (zerop err)
            (signal-mac-os-error err))))
      ;; Compute the color map for n-bit grayscale
      (let* ((n-colors (ash 1 bits-per-pixel))
             (shift (- 16 bits-per-pixel))
             (rgb (make-rgbcolor))
             (r (make-rect :left 0 :top 0 :right width :bottom height))
             (the-port (_getport)))
        (with-temps ((palette (lambda ()
                                (_newpalette n-colors
                                             0 (cconstant pmTolerant) 0))
                              _disposepalette))
          (loop for i from 0 below n-colors do
            (let ((k (lognot (ash i shift))))
              (setf (rgbcolor-red rgb 0) (ldb (byte 16 0) k))
              (setf (rgbcolor-green rgb 0) (round (* .9 (ldb (byte 16 0) k))))
              (setf (rgbcolor-blue rgb 0) (round (* .9 (ldb (byte 16 0) k))))
              (_setentrycolor palette i rgb)))
          (with-temps ((pmh _newpixmap _dispospixmap))
            (with-handle-locked (data)
              (with-handle-locked (pmh)
                ;; This could have used a bunch of (setf (octet-ref-remote-handle ...)),
                ;; but since several fields are being altered, it seemed better to
                ;; drag the whole thing over to Lisp, fiddle with it, and send it back.
                ;; A pixMap is only 50 bytes anyway.
                (with-mac-struct (pixmap pixmap :handle pmh)
                  (setf (pixmap-baseaddr pixmap 0) (_ptrfromhandle data))
                  (setf (pixmap-rowbytes pixmap 0)
                        (logior #x8000
```

```
                                ;; See Inside Macintosh p. V-53.
                                (/ (* width bits-per-pixel) 8)))
                  (setf (pixmap-bounds pixmap 0) r)
                  (setf (pixmap-pixelsize pixmap 0) bits-per-pixel)
                  (setf (pixmap-cmpsize pixmap 0) bits-per-pixel)
                  (_write-opaque-bytes-into-handle
                    pmh (length pixmap) pixmap))
                ;; Now don't ask me why we bring it back again.  Maybe it's because
                ;; drawpixmap insists on having a Lisp pixMap to transport back to
                ;; the Macintosh side.
                (with-mac-struct (pixmap pixmap :handle pmh)
                  (_palette2ctab palette (pixmap-pmtable pixmap 0))
                  (let ((portrect (octet-ref-remote-ptr (grafport-portrect the-port)))
                        (portbits (octet-ref-remote-ptr (grafport-portbits the-port))))
                    (let ((pict (_openpicture r)))
                      (unless (typep portbits '(vector (unsigned-byte 8)))
                        (error "Bad portbits"))
                      (with-clip-rect (r)
                        (_drawpixmap pixmap portrect portrect
                                     (cconstant srcCopy) 0))
                      (_closepicture)
                      pict))))))))

;;;=======================================
;;; An attempt at making this toy a little more interesting
;;;=======================================

;;---The menu item for this should be disabled when there are no pictures.
(define-pic-show-command (com-set-zoom :menu-accelerator "Set Zoom"
                                       ;; I think the -c- gets lost somehow, which
                                       ;; puts this on the same key as [Edit / Undo],
                                       ;; which fails to adhere to the Macintosh UI guidelines.
                                       :keyboard-accelerator #\s-c-Z
                                       :menu-level (:top-level (:mac "PicShow")))
    ((zoom 'number :default 1 :prompt "Zoom"))
   (let ((wp (front-window-picture self)))
     (if wp
         (set-window-picture-zoom wp zoom)
         (beep))))

(defmethod (front-window-picture pic-show) ()
  (let ((mac-window (_frontwindow)))
    ;; Look up the front Macintosh window in our application data structures.
    (let ((window-picture (find mac-window window-pictures :key #'window-picture-mac-window)))
      window-picture)))
```

```
(defmethod (set-window-picture-zoom window-picture) (new-zoom)
  (let ((old-zoom zoom))
    (setq zoom new-zoom)
    (with-qd-port (mac-window)
      ;; By invalidating the old and new sizes, we hope to cause correct redisplay
      (_invalrect display-rect)
      ;; Compute new edges
      (let ((left (round (* zoom (rect-left pict-rect 0))))
            (top (round (* zoom (rect-top pict-rect 0))))
            (right (round (* zoom (rect-right pict-rect 0))))
            (bottom (round (* zoom (rect-bottom pict-rect 0)))))
        ;; Update scroll bars
        (flet ((foo (s v)
                 (let ((old (_getctlvalue s)))
                   ;; Set the max before the value -- if we did the value first,
                   ;; it might get clipped to the max.
                   (_setctlmax s v)
                   (_setctlvalue s (round old (/ zoom old-zoom))))))
          (foo (first scroll-bars) (- right left))
          (foo (second scroll-bars) (- bottom top)))
        ;; Update the stored version
        (setf (rect-left display-rect 0) (round left))
        (setf (rect-top display-rect 0) (round top))
        (setf (rect-right display-rect 0) (round right))
        (setf (rect-bottom display-rect 0) (round bottom))
        (_invalrect display-rect)))))

;;;=======================================
;;; Ideas for enhancements
;;;=======================================
#|
There should be a Windows command which offers a menu of open picture windows,
and when you pick one, selects the corresponding Macintosh window.

There could be a Crop command which lets you pick a rectangle and reduce
the image to only that size.

There could be keyboard accelerators to zoom and unzoom (and scroll, too) without
messing around with the dialog.

The _openpicture/_closepicture stuff should be macroized into making-picture

|#

(compile-flavor-methods pic-show window-picture)
```

Menubar Example

menubar.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MAC-TOOLBOX; Base: 10 -*-
;;;> EXAMPLES-MESSAGE
;;;>
;;;>*****************************************************************************************
;;;>
;;;>       Symbolics hereby grants permission to customers to incorporate
;;;>       the examples in this file in any work belonging to customers.
;;;>
;;;>*****************************************************************************************

;;;
;;; You want to run show-menu-bar-menus.
;;; Also look at menu-item-feature and its setf method.
;;;

;;;=======================================
;;; Structure definitions from Inside Macintosh V-229
;;;=======================================

(define-octet-structure (MenuRec :access-type :byte-swapped-8)
  (menuOH menuhandle)
  (menuLeft integer-16))

(define-octet-structure (HMenuRec :access-type :byte-swapped-8)
  (menuHOH menuhandle)
  (* integer-16))

;; field types too
(define-octet-structure-field-type MenuRec () (array index)
  :result-type T
  :size (octet-structure-total-size (MenuRec .x. 0))
  :data `(macintosh-internals::make-included-octet-structure MenuRec ,array ,index))

(define-octet-structure-field-type HMenuRec () (array index)
  :result-type T
  :size (octet-structure-total-size (HMenuRec .x. 0))
  :data `(macintosh-internals::make-included-octet-structure HMenuRec ,array ,index))
```

```
(define-octet-structure (DynamicMenuList :access-type :byte-swapped-8)
  (lastMenu cardinal-16)
  (lastRight cardinal-16)
  (mbResID cardinal-16)
  (menu (vector MenuRec (/ lastMenu 6)))
  (lastHMenu cardinal-16)
  (menuTitleSave pixmaphandle)
  (hMenu (vector HMenuRec (/ lastHMenu 6)))
  )

;; This is the anonymous internal structure in IM V-230
(define-octet-structure (menu-item :access-type :byte-swapped-8)
  (itemStringLength cardinal-8)
  (itemString (vector character-8 itemStringLength))
  (itemIcon cardinal-8)
  (itemCmd cardinal-8)                          ;it's not always character-8
  (itemMark cardinal-8)                         ;it's not always character-8
  (itemStyle Style)
  )

(defmacro string-from-str255-reference (reference)
  (declare (values string next-index))
  `(string-from-str255
     ,(second reference)
     (octet-structure-field-index ,reference)))

(defun string-from-str255 (octet-array index &key area)
  (declare (values string next-index))
  (let ((len (aref octet-array index)))
    (let ((result (if (eql area :stack)
                      (sys:make-stack-array len :element-type 'string-char)
                      (make-string len :area area)))
          (octet-array octet-array))
      (declare (sys:array-register result octet-array))
      (dotimes (i len)
        (setf (aref result i) (code-char (aref octet-array (+ index i 1)))))
      (values result (+ index len 1)))))

(defmacro string=-str255-reference (string reference)
  `(string=-str255-reference-1 ,string ,(second reference)
                               (octet-structure-field-index ,reference)))
```

```
(defun string=-str255-reference-1 (string octet-array index)
  (let ((len (length string)))
    (and (= len (aref octet-array index))
         (let ((string string) (octet-array octet-array))
           (declare (sys:array-register string octet-array))
           (dotimes (i len T)
             (unless (char= (aref string i) (code-char (aref octet-array (+ index i 1))))
               (return nil)))))))

;;;=======================================
;;; The application code
;;;=======================================

;; This is the thing to run
(defun show-menu-bar-menus ()
  (with-temps ((menu-bar-handle _getmenubar _disposhandle))
    (with-mac-struct (menu-bar () :handle menu-bar-handle)
      (format T "~&There are ~d items, right edge ~d"
              (/ (dynamicmenulist-lastmenu menu-bar 0) 6)
              (dynamicmenulist-lastright menu-bar 0))
      (let ((mbdf (dynamicmenulist-mbResID menu-bar 0)))
        (when (≠ mbdf 0)
          (format T "~&Drawn by MBDF resid ~d, variant ~d"
                  (ldb (byte 13 3) mbdf)
                  (ldb (byte 3 0) mbdf))))
      (domap () ((item (dynamicmenulist-menu menu-bar 0)))
        (with-mac-struct (menuinfo () :handle (menurec-menuoh item 0))
          (show-menuinfo menuinfo
                         (menurec-menuleft item 0))))
      (format T "~&There are ~d hierarchical/popup menu items"
              (/ (dynamicmenulist-lastHmenu menu-bar 0) 6))
      (domap () ((item (dynamicmenulist-hmenu menu-bar 0)))
        (with-mac-struct (hmenuinfo () :handle (hmenurec-menuhoh item 0))
          (show-menuinfo hmenuinfo ()))))))
```

```
(defun show-menuinfo (menuinfo left)
  (format T "~&  Menu ID ~d, size ~dw*~dh"
          (menuinfo-menuid menuinfo 0)
          (menuinfo-menuwidth menuinfo 0)
          (menuinfo-menuheight menuinfo 0)
          )
  (when left (format T ", left ~d" left))
  (let ((def (menuinfo-menuProc menuinfo 0)))
    (when (≠ def 0)
      (format T ", defProc handle 0x~x" def)))
  (let ((enables (ldb (byte 32 0) (menuinfo-enableFlags menuinfo 0))))
    (multiple-value-bind (title next-index)
        (string-from-str255-reference (menuinfo-menudata menuinfo 0))
      (format T "~&    Title ~s" title)
      (unless (ldb-test (byte 1 0) enables) (format T " [menu disabled]"))
      (let ((index next-index))
        (loop for i from 1 do
          (multiple-value-bind (item-text icon-number kbd-equivalent
                                checkmark-char char-style next-index)
              (menu-item-stuff menuinfo index)
            (when (null item-text) (return))
            (format T "~&      Item ~s" item-text)
            (when (≠ icon-number 0)
              (format T ", icon ~d" (+ 256 icon-number)))
            (unless (eql kbd-equivalent 0)
              (cond ((eql kbd-equivalent #x1B)
                     ;; hierarchical menu, see IM V-236
                     (format T ", submenu ~d" checkmark-char))
                    (T
                     (if (≤ #x1B kbd-equivalent #x1F)
                         (format T ", kbd 0x~x" kbd-equivalent)
                         (format T ", kbd ~c" kbd-equivalent))
                     (unless (eql checkmark-char 0)
                       (format T ", checkmark ~c" checkmark-char)))))
            (when (≠ char-style 0)
              ;; --- decode bits of the enum Style
              (format T ", style ~d" char-style))
            (unless (ldb-test (byte 1 i) enables)
              (format T " [disabled]"))
            (setq index next-index)))
        (unless (= index (length menuinfo))
          (format T "~&    Extra ~d bytes" (- (length menuinfo) index)))))))
```

```
(defun menu-item-stuff (octet-array index)
  (declare (values item-text icon-number kbd-equivalent checkmark-char char-style next-index))
  (when (> (menu-item-itemStringLength octet-array index) 0)
    (values (menu-item-itemString octet-array index)
            (menu-item-itemIcon octet-array index)
            (menu-item-itemCmd octet-array index)
            (menu-item-itemMark octet-array index)
            (menu-item-itemStyle octet-array index)
            (octet-structure-total-size
              (menu-item octet-array index)))))

;;;=======================================
;;; Here's a little more
;;;=======================================

;;; Pass nil or the-item-text to refer the the menu itself
(defun menu-item-feature (menu-text item-text feature)
  (check-type feature (member :text :enabled :checked :mark :icon :style :key))
  (multiple-value-bind (handle item-number)
      (menu-handle-and-item-number menu-text item-text)
    (ecase feature
      (:text
        (with-str255 (string)
          (_GetItem handle item-number string)
          (copy-seq string)))
      (:enabled
        (if (< item-number 32)
            (with-mac-struct (menuinfo () :handle handle)
              (ldb-test (byte 1 item-number) (menuinfo-enableFlags menuinfo 0)))
            T))
      (:checked
        (not (zerop (_getitemmark handle item-number))))
      (:mark
        (let ((number (_getitemmark handle item-number)))
          (if (zerop number) nil (code-char number))))
      (:icon
        (let ((n (_getitemicon handle item-number)))
          (if (zerop n) nil (+ n 256))))
      (:style
        ;; could decode the bits here
        (_getitemstyle handle item-number))
      (:key
        (_getitemcmd handle item-number))
      )))
```

```
;; Internal, for use by setf below
(defun set-menu-item-feature (menu-text item-text feature value)
  (check-type feature (member :text :enabled :checked :mark :icon :style :key))
  (multiple-value-bind (handle item-number)
      (menu-handle-and-item-number menu-text item-text)
    (ecase feature
      (:text
        (_SetItem handle item-number value))
      (:enabled
        (if (< item-number 32)
            (if value
                (_enableitem handle item-number)
                (_disableitem handle item-number))
            (check-type value (not null))))
      (:checked
        (_checkitem handle item-number value))
      (:mark
        (_setitemmark handle item-number (if value (char-code value) 0)))
      (:icon
        (check-type value (or null (integer 256 (512))))
        (_setitemicon handle item-number (if value (- value 256) 0)))
      (:style
        ;; could encode the bits here
        (check-type value (integer 0 (256)))
        (_setitemstyle handle item-number value))
      (:key
        (_setitemcmd handle item-number (if value (char-code value) 0)))
      )))

(defsetf menu-item-feature set-menu-item-feature)
```

```
;; The thing that finds the handle and item number
(defun menu-handle-and-item-number (the-menu-text the-item-text &key (if-not-found :error))
  (block found
    (flet ((check-menuinfo (handle)
             (with-mac-struct (menuinfo () :handle handle)
               (when (string=-str255-reference the-menu-text (menuinfo-menudata menuinfo 0))
                 (when (null the-item-text)
                   (return-from found (values handle 0)))
                 (let ((next-index (let ((i (octet-structure-field-index
                                             (menuinfo-menudata menuinfo 0))))
                                     (+ i 1 (aref menuinfo i)))))
                   (let ((index next-index))
                     (loop for i from 1 do      ;it -IS- 1-origin, right?
                       (multiple-value-bind (item-text icon-number kbd-equivalent
                                             checkmark-char char-style next-index)
                           (menu-item-stuff menuinfo index)
                         (ignore icon-number kbd-equivalent checkmark-char char-style)
                         (when (null item-text) (return))
                         (when (string= item-text the-item-text)
                           (return-from found
                             (values handle i)))
                         (setq index next-index)))))))))
      (with-temps ((menu-bar-handle _getmenubar _disposhandle))
        (with-mac-struct (menu-bar () :handle menu-bar-handle)
          (domap () ((item (dynamicmenulist-menu menu-bar 0)))
            (check-menuinfo (menurec-menuoh item 0)))
          (domap () ((item (dynamicmenulist-hmenu menu-bar 0)))
            (check-menuinfo (hmenurec-menuhoh item 0))))))
    (ecase if-not-found
      (:error (error "Failed to find Macintosh menu ~s item ~s" the-menu-text the-item-text))
      ((nil) ))
    ))

#||
;;Here are some trivial things you can do with the Genera menu bar

;; A Keyboard accelerator to get at the Keyboard Kontrol
(setf (menu-item-feature "Keyboard" "Keyboard Control" :key) #\K)

;; Call the finder with a single keystroke!  (clover-F)
(setf (menu-item-feature "∀" "Finder" :key) #\F)
;; MacOS seems to use the keystroke internally -- restore with this
(setf (menu-item-feature "∀" "Finder" :key) (code-char 29))

(setf (menu-item-feature "Options" "Keyboard" :style)
      (+ (cconstant bold) (cconstant shadow)))
```

```
||#
```

Show-icons Example

show-icons.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MAC-TOOLBOX; Base: 10 -*-
;;;> EXAMPLES-MESSAGE
;;;>
;;;>*****************************************************************************************
;;;>
;;;>       Symbolics hereby grants permission to customers to incorporate
;;;>       the examples in this file in any work belonging to customers.
;;;>
;;;>*****************************************************************************************

(defun show-all-mac-cursors ()
  (do-rsrcs (curs ignore ignore) ("CURS" :all :load T)
    (draw-mac-cursor-at-cursorpos :handle curs))
  (send *standard-output* :increment-cursorpos 0 16))

(defun show-all-mac-icons ()
  (do-rsrcs (h id name) ("ICON" :all :load T)
    (format T "~%ICON ~d ~s~%" id name)
    (draw-mac-icon-at-cursorpos :handle h)
    (send *standard-output* :increment-cursorpos 0 32))
  (do-rsrcs (h id name) ("ICN#" :all :load T)
    (format T "~%ICN# ~d ~s~%" id name)
    (with-mac-struct (icns () :handle h)
      (let ((icon-length (/ (* 32 32) 8)))
        (loop for i from 0 below (length icns) by icon-length do
          (stack-let ((icon (make-array icon-length :element-type '(unsigned-byte 8)
                                        :displaced-to icns
                                        :displaced-index-offset i)))
            (draw-mac-icon-at-cursorpos :struct icon)))))
    (send *standard-output* :increment-cursorpos 0 32)))
```

```
(defun draw-mac-icon-at-cursorpos (&key resnum handle struct)
  (cond (struct
         (stack-let* ((struct struct)
                      (image (make-array '(32 32) :element-type 'bit))
                      (image8 (make-array (/ (* 32 32) 8)
                                          :element-type '(unsigned-byte 8)
                                          :displaced-to image)))
           (declare (sys:array-register struct image8))
           (dotimes (j 128)
             (setf (aref image8 j) (sys:bit-reverse-8 (aref struct j))))
           (multiple-value-bind (x y) (send *standard-output* :read-cursorpos)
             (graphics:draw-image image x y)
             (send *standard-output* :increment-cursorpos 36 0))))
        (resnum
         (with-resource (icon "ICON" resnum)
           (draw-mac-icon-at-cursorpos :struct icon)))
        (handle
         (with-mac-struct (icon () :handle handle)
           (draw-mac-icon-at-cursorpos :struct icon)))
        (T )))

(defun draw-mac-cursor-at-cursorpos (&key resnum handle struct)
  (cond (struct
         (check-type struct (vector (unsigned-byte 8) 68))
         (stack-let* ((image (make-array '(32 32) :element-type 'bit))
                      (image8 (make-array 64 :element-type '(unsigned-byte 8)
                                          :displaced-to image)))
           (let ((i 0) (j 0))
             (dotimes (row 16)
               (dotimes (col 2)
                 (setf (aref image8 j) (sys:bit-reverse-8 (aref struct i)))
                 (incf i)
                 (incf j))
               (incf j 2)))
           (multiple-value-bind (x y) (send *standard-output* :read-cursorpos)
             (graphics:draw-image image x y)
             (send *standard-output* :increment-cursorpos 20 0))))
        (resnum
         (with-resource (struct "CURS" resnum)
           (draw-mac-cursor-at-cursorpos :struct struct)))
        (handle
         (with-mac-struct (struct () :handle handle)
           (draw-mac-cursor-at-cursorpos :struct struct)))
        (T )))
```

MacIvory Interface to HyperCard

The HyperIvory Demo

What is HyperIvory?

HyperIvory is a collection of demos that illustrates the close communication and integration possible between Ivory and a host HyperCard application. A similar approach could be used with any host application that allows extension, not just HyperCard. This kind of integration and communication can also be built into new applications, as described in the "MacIvory Programmer's Reference".

HyperIvory offers examples of how to do the extension. The HyperIvory stack illustrates some of what Ivory could bring to HyperCard, including:

- Simple Lisp, as used in the EVAL Service and Command Service
- Programs, such as the Stack Dumper and Iconizer
- Hypertext capabilities, shown in a static form in the Doc Ex demo, which provides an interface to the same doc that Document Examiner gets you; and shown in a computed form in the Flavor Demo, which uses Genera Flavor components and dependents.
- Statice, as used in the database for Mug Shots.
- Joshua/ES, as used in Map Routes.

The Lisp code for these demos is in the directory SYS:EMBEDDING;MACIVORY;HYPERCARD;.

To use these demos:

1. Load the system HyperCard/MacIvory into Genera before using the HyperIvory demos.
2. Double-click on the HyperIvory icon, located in the MacIvory Applications folder, to activate the stack.

You do not need to run the Genera icon when you run the HyperIvory stack. The HyperIvory stack will boot your Ivory if it is not booted. Or, you can run Genera to boot and set up the software, then use [File / Quit] to get out of Genera and then start up HyperCard. Sometimes it's useful to run Genera and HyperCard at the same time (using MultiFinder), so you can use Genera to debug your XFCN servers.

The demo stack consists of these cards:

HyperIvory Apologia
: This card explains the general purpose of HyperIvory and presents some overall information about the cards.

HyperIvory menu	This card is a menu of the available demos.
IvoryCom	This card provides low-level interface directly to the CallIvory XFCN.
Stack Dumper	The Stack Dumper card dumps much of a HyperCard stack into a text file, using a Lisp-readable syntax. You should probably read the code if you anticipate using this seriously.
EVAL Service	EVAL Service evaluates Lisp forms you type in.
Ivory At Your Command	You can issue and activate Genera Command Processor commands with from this card.
The Obligatory Flavor Demo	This card returns the dependents and components of a flavor you specify. This demo goes beyond ordinary hypertext in that it computes dynamically variable results rather than linking to prewritten text.
Spellbound	A spelling checker. See the section "Example of Creating a CallIvory Server".
The Hypertext Delivery Story	This card provides an interface to the Genera Document Examiner.
Iconizer	Uses Lisp code to scan the Finder Desktop to extract icons and paste them onto cards.
Map	This demo computes and displays routes among various towns.
Mug Shots	This demo illustrates the integration of HyperCard with Statice. It displays "mug shots" from a Statice database.

What's in it?

HyperIvory consists of a HyperCard stack and some Lisp files. In the stack itself are:

- An XFCN so HyperTalk scripts can talk to Ivory.
- Some HyperTalk scripting so HyperTalk scripts can understand Ivory's reply.
- Several cards with their own buttons, scripts, and so on.

In Genera itself:

- General code for fielding HyperCard requests and for calling back into HyperCard.

- Application-specific code to handle requests from the scripts of specific cards and their buttons.
- HyperCard-specific "glue" code.

How does it work?

The XFCN is the heart. It provides a HyperTalk function `CallIvory(`*routineName, argsString*`) -> resultString`. This uses the RPC mechanism to pass the request to Ivory. The RPC server looks up the Lisp XFCN server corresponding to *routineName*, and calls it, passing it the *argsString*. The values returned by the server routine, whatever it types to ***standard-output***, and any errors, are returned in `resultString`.

While the XFCN is awaiting the server's response, it provides RPC service to Ivory — including, notably, a path for calling back into HyperCard. This enables Lisp code to do virtually anything within HyperCard that a HyperTalk script could do.

You may wish to consult the *Macintosh HyperCard User's Guide* or *Macintosh HyperCard Script Language Guide: the HyperTalk Language* for detailed documentation on these topics.

HyperTalk Routines

The XFCN "CallIvory" is attached to the HyperIvory stack as for any XFCN. The result extractors are in the script of the HyperIvory stack. See the section "Adding a CallIvory Server" for an example of using these routines.

MacIvory includes these new HyperTalk routines for calling Ivory and for extracting from the results returned by calling Ivory:

CallIvory HyperTalk Routine

HyperTalk Syntax: `put CallIvory(`*routineName, routineArgs*`) into` *results*

Description: Calls Ivory's XFCN server named *routineName*, passing it the string *routineArgs*. Whatever that server routine returns is put into the HyperCard container *results*.

results receives in a single string the description of the Lisp error that occurred, if any; everything the server typed to its ***standard-output***; and each value returned by the server. Use the HyperTalk functions ResultError, ResultTypeout, ResultValues, ResultNValues, ResultValue to extract portions of the returned value. If a Lisp value returned is a string, that string is returned to HyperCard; otherwise its **write-to-string** is returned.

ResultError HyperTalk Routine

HyperTalk Syntax: `put ResultError(`*results*`) into xxx`

Description: Extracts the text of a Lisp error, if one occurred while the server ran. Otherwise, leaves this empty.

ResultTypeout HyperTalk Routine

HyperTalk Syntax: `put ResultTypeout(`*results*`) into xxx`

Description: Extracts server typeout to ***standard-output***, if any.

ResultValues HyperTalk Routine

HyperTalk Syntax: `put ResultValues(`*results*`) into xxx`

Description: Extracts all Lisp values returned by the server, separated by return characters.

ResultNValues HyperTalk Routine

HyperTalk Syntax: `put ResultNValues(`*results*`) into xxx`

Description: Extracts the number of Lisp values returned by the server.

ResultValue HyperTalk Routine

HyperTalk Syntax: `put ResultValue(`*result, valueNumber*`) into xxx`

Description: Extracts a specific Lisp value returned by the server. The first value is valueNumber 1.

Lisp Routines

In addition to the Lisp functions and macros described here, you can also use all the Macintosh toolbox routines listed in "Lisp Functions That Access the Macintosh Toolbox".

mtb:define-xfcn-server *name arglist &body body* *Macro*

Defines a XFCN server *name* callable by HyperCard. Its *arglist* should be of length one, receiving the string passed by the invoking call to the HyperTalk XFCN CallIvory.

The server is run within a **condition-bind** for **error**; princ-to-string of the error is returned to HyperCard. In some circumstances it may be desirable to debug problems using the Debugger. Set ***debug-hc-server-flag*** to T to disable the condition-bind.

See the section "Hints for Writing CallIvory Servers" for related information".

Functions Useful to Call from XFCN Servers

mtb:hc-chunk-character *string from &optional to* *Function*

Returns a substring of *string* from character *from* to *to*, or the one character at *from*. Similar to HyperTalk's *character* chunk expression, except **mtb:hc-chunk-character** is zero-based. This function is useful for destructuring the string argument passed to XFCN server routines.

mtb:hc-chunk-item *string from &optional to* *Function*

Returns a substring of *string* containing items *from* to *to*, or the one item *from*. Similar to HyperTalk's *item* chunk expression, except **mtb:hc-chunk-item** is zero-based. Items are separated by commas. This function is useful for destructuring the string argument passed to XFCN server routines.

Storage Allocation

mtb:with-hc-zstring *(var string) &body body* *Macro*

Allocates a Macintosh handle to hold a zstring copy of the Lisp string *string*, and binds the Lisp variable *var* to that handle, deallocating it when *body* exits. A HyperCard zstring is terminated by a zero byte, so don't send down Lisp bullet characters. The resulting handle can be passed to the several HyperCard callbacks that want zstring handles, such as **mtb:hc-set-global**.

mtb:with-hc-eval-expr *(result string) &body body* *Macro*

Binds *result* to the HyperCard evaluation of the HyperCard expression *string*, for the duration of *body*. This sends *string* to HyperCard, sends *result* back to Lisp, and deallocates the HyperCard handle when done.

mtb:making-qd-picture *(rect) &body body* *Macro*

Returns a Macintosh handle to a QuickDraw picture created by Lisp calls to QuickDraw within *body*. (See the section "Lisp Functions That Interface the Macintosh QuickDraw Manager" for a complete list of these functions.) Uses the

QuickDraw operations OpenPicture and ClosePicture around *body*. The picture's rect is *rect*.

For example:

```
(defun make-pict-file-from-icon (icon-handle pathname)
 (with-open-refnum (out pathname :permission (cconstant fsWrPerm))
  (_SetEOF out 512)
  (let ((pict (with-rect (r 0 0 32 32)
                  (making-qd-picture (r) (_ploticon r handle)))))
   (let ((err (with-handle-locked (pict)
                  (_fswrite-remote out (_ptrfromhandle pict) (_gethandlesize pict)))))
    (unless (zerop err)
     (mtb::signal-mac-os-error err))))))
```

Callbacks to HyperCard: MacIvory Glue Functions

This section describes the MacIvory glue routines, which are used in making your external commands communicate with HyperCard. These "glue" routines, or callbacks, form an interface through which you can access HyperCard's internal routines. The argument list for each routine describes Lisp and HyperCard parameters. HyperCard parameters are set in their own typeface. For example, in the argument list *msg*(`string`), *msg* is a Lisp value and `string` is a Macintosh value.

Note that not all callbacks defined by HyperCard are implemented with remote entries. The simplest of string operations, coercions, and format conversions are amply handled by Symbolics Common Lisp; there is no need to call back to HyperCard. These are the unimplemented routines:

```
BoolToStr
ExtToStr
LongToStr
NumToHex
NumToStr
PasToZero
ReturnToPas
ScanToReturn
ScanToZero
StringEqual
StringLength
StringMatch
StrToBool
StrToExt
StrToLong
StrToNum
ZeroToPas
```

mtb:hc-send-card-message *msg(*`string`*)* *Function*

Sends a HyperCard message to the current card. Compare with **mtb:hc-send-hc-message**. HyperCard also contrasts expression evaluation with message sending.

mtb:hc-send-hc-message *msg(*`string`*)* *Function*

Sends a HyperCard message, bypassing the normal inheritance path. Compare with **mtb:hc-send-card-message**. HyperCard also contrasts expression evaluation with message sending.

mtb:hc-eval-expr *expr(*`string`*)* *Function*

HyperCard evaluates the expression and allocates a handle to hold its result. You probably really want to use **mtb:with-hc-eval-expr**, which checks the errcode and deallocates the handle for you. HyperCard contrasts expression evaluation with message sending.

Returns result(handle), errcode(integer-16).

mtb:hc-zero-bytes *address(*`Ptr`*), length(*`integer-32`*)* *Function*

Zeros *length* bytes of Macintosh memory starting at *address*.

mtb:hc-get-global *global-name(*`string`*)* *Function*

Gets the value of HyperCard global variable. Note that you must deallocate the handle.

Returns result(handle).

mtb:hc-set-global *global-name(*`string`*), value(*`Handle`*)* *Function*

Sets the value of HyperCard global variable. Note that this callback does not place the value into the handle for you.

mtb:hc-get-field-by-name *cardp(*`Boolean`*), field-name(*`string`*)* *Function*

Gets the contents of the named text field (card or background, depending on cardp). Note that you must deallocate the handle.

Returns result(handle).

mtb:hc-get-field-by-num *cardp(*`Boolean`*), number(*`integer-16`*)* *Function*

Gets the contents of the indexed text field (card or background, depending on the value of cardp). Note that you must deallocate the handle.

Returns result(handle).

mtb:hc-get-field-by-id *cardp(*Boolean*), id(*integer-16*)* *Function*

Gets the contents of the identified text field (card or background, depending on cardp). Note that you must deallocate the handle.

Returns result(handle).

mtb:hc-set-field-by-name *cardp(*Boolean*), name(*string*), value(*Handle*)* *Function*

Sets the contents of the named text field (card or background, depending on the value of cardp). Note that this callback does not place the value into the handle for you.

mtb:hc-set-field-by-num *cardp(*Boolean*), number(*integer-16*), value(*Handle*)* *Function*

Sets the contents of the indexed text field (card or background, depending on the value of cardp). Note that this callback does not place the value into the handle for you.

mtb:hc-set-field-by-id *cardp(*Boolean*), id(*integer-16*), value(*Handle*)* *Function*

Sets the contents of the identified text field (card or background, depending on cardp). Note that this callback does not place the value into the handle for you.

Existing CallIvory Servers

mtb:list-routines *ignore* *Function*

Prints a comma-separated list of the names of all CallIvory servers.

mtb:echo-arg *string* *Function*

Returns *string*.

mtb:beep-some *count* *Function*

Beeps *count* times.

mtb:eval-some *forms* *Function*

Reads forms from the string *forms*, and **eval**s them. The first value from each form becomes a value of **mtb:eval-some**.

mtb:command-some *command-and-args* *Function*

Runs the Command Processor command line *command-and-args*. The following CallIvory servers implement or support the behavior of the corresponding cards:

flavor-parts
cards-for-all-desktop-icons
cards-for-sage-topic
map-hack-click
list-mugshots-starting-with
get-mugshot-for
get-mugshot-for-1

Hints for Writing CallIvory Servers

- Note that if HyperCard is stuck, there is no UI service for Genera. If the XFCN server is in the Debugger, HyperCard is stuck. If there is no UI service for Genera, you cannot type out the Debugger. Contact the Symbolics Consulting Group if you need assistance with this.

- Other RPC agents, besides EMB, work, if built into the XFCN.

- Remember to deallocate the handles HyperCard allocates for you. If you gradually run out of HyperCard heap space, the HyperTalk function HeapSpace might help. In some cases HyperCard itself uses heavy dynamic memory allocation/deallocation.

Example of Creating a CallIvory Server

This simple example shows the process of adding a new CallIvory server for Lisp to respond to requests initiated by HyperCard. The code for the Lisp server form, along with four HyperTalk scripts, is distributed in the file SYS:EMBEDDING;MACIVORY;HYPERCARD;SPELL.LISP.

This discussion assumes that you are familiar with using and programming both Genera and HyperCard.

The distributed HyperIvory stack and the associated HyperCard/MacIvory system already contain the code discussed here, so you may have to ignore various redefinition warnings. See Figure 97 with the "Spellbound" card in the HyperIvory stack.

You must inform HyperCard that you are authorized to write the HyperTalk scripts in this example. To do this:

1. While running HyperCard, go to the Home card (choose the "Home" menu command).

2. Go to the User Preferences card (choose the "Prev" menu command).

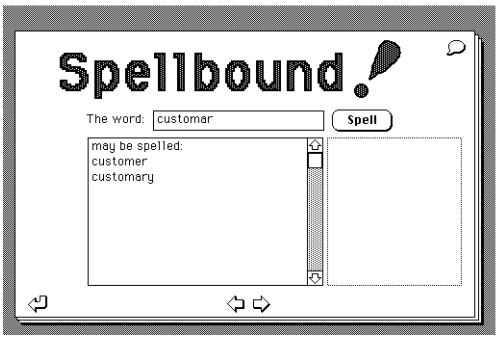

Figure 97. The "Spellbound" card

3. Click on the "Scripting" radio button.

Now:

1. Compile the define-xfcn-server form.

   In Genera, switch to Zmacs and compile this code. It is distributed in SYS:EMBEDDING;MACIVORY;HYPERCARD;SPELL.LISP.

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(mtb::define-xfcn-server spell-word (word)
  (setq word (string-trim '(#\space #\return #\tab) word))
  (if (zwei:word-in-dictionaries-p word (string-length word))
      (write-line "is spelled correctly.")
      (let ((corrections (zwei:get-all-corrections word)))
        (cond ((null corrections)
               (write-line "is unknown and possibly misspelled."))
              (T
               (write-line "may be spelled:")
               (mapc #'write-line corrections))))))
```

   This form returns its results to HyperCard by printing to ***standard-output***. The HyperTalk script receiving the value simply puts it into a text field.

2. Find an appropriate HyperCard stack.

   You have several choices:

   - Simply use the distributed HyperIvory stack, duplicate that stack using the Finder, and modify the copy.

- Open the HyperIvory stack, use HyperCard's "Save a Copy ..." menu item, and modify the copy.

- Use HyperCard to make a new stack, and manually move the stack script (using HyperCard) and the XFCN resource (using ResEdit or equivalent).

3. Create a card.

   In HyperCard, use the "New Card" menu command. The new card will share its background with the background of the card you're looking at.

4. Create fields "Word", "Respellings", "Errors".

   Use the "New Field" menu command. Drag the fields around to place and size them the way you want. Use the "Field Info" menu command to set the field names, which are referred to by the HyperTalk scripts.

   While you have the "Word" field open, type in its script.

```
--This is just so you don't have to move your hand from kbd to mouse
on returnInField
  send closeField to me
  spellTheWord
end returnInField
```

5. Create button "Spell".

   Use the "New Button" menu command and the "Button Info" menu command.

   While you have the "Spell" button open, type in its script.

```
on mouseUp
  spellTheWord
end mouseUp
```

6. Add script for card.

   Use the "Card Info" menu command to get to the card's script window, and type it in. If you are tired of typing, and you are running MultiFinder, you could take advantage of the communication between Genera kill-ring and Macintosh clipboard:

   a. Switch from HyperCard to Genera.

   b. Select Zmacs and get the example code in a buffer.

   c. Mark the code and use the "Copy" command from the menu bar.

   d. Switch back to HyperCard.

e. Get to the card's script window.

f. Choose the "Paste" menu command.

```
on openCard
  --start off with no errors showing
  hide card field "Errors"
  --be able to start typing without selecting/killing first
  select text of card field "word"
  pass openCard
end openCard

on spellTheWord
  put empty into card field "Respellings"
  hide card field "Errors"
  --Hilight the Spell button to provide feedback to user while doing the work
  set the hilite of card button "Spell" to true
  --do the real work
  put CallIvory("spell-word", card field "word") into results
  --check for errors, and report them
  put ResultError(results) into card field "Errors"
  if card field "Errors" is not empty then
    put "Lisp Error:" & return before card field "Errors"
    show card field "Errors"
  else
    --unscroll from last time, perhaps
    set the scroll of card field "Respellings" to 0
    --show respellings if any
    put ResultTypeout(results) into card field "Respellings"
  end if
  --Done with work, unhilight
  set the hilite of card button "Spell" to false
end spellTheWord
```

7. Draw some glitzy titles and so on.

   Scribble all over the card if it suits your purposes.

8. Link to the new card with some buttons somewhere.

   This is optional, of course.

Lisp Functions That Access the Macintosh Toolbox

The following tables list the Lisp functions that let you access the corresponding routines in the Macintosh Toolbox. The routines are grouped by Apple managers and listed alphabetically. The tables show you the name of a Lisp function, its arguments, and any returned values. All Lisp functions are preceded by the package prefix **mtb:**. These functions provide automatic (Macintosh) type translation.

Note that names are given in Macintosh case style for readability, but that, in fact, these functions are case insensitive.

Adb Manager

Lisp Function	*Arguments*	*Values Returned*
_ADBOp	(*data compout buffer commandnum*)	
_ADBReInit	()	
_CountADBs	()	(*devicecount*)
_GetADBInfo	(*infoin address*)	(*infoout*)
_GetIndADB	(*infoin devtableindex*)	(*address infoout*)
_SetADBInfo	(*info address*)	

Appletalk-Manager

No Entries defined for Appletalk-Manager.

Color-Manager

Lisp Function	*Arguments*	*Values Returned*
_AddComp	(*compproc*)	
_AddSearch	(*searchproc*)	
_Color2Index	(*rgb*)	(*index*)
_DelComp	(*compproc*)	
_DelSearch	(*searchproc*)	
_GetCTSeed	()	(*seed*)
_GetSubTable	(*mycolor* ITABRES *targettbl*)	
_Index2Color	(*index rgbin*)	(*rgbout*)
_InvertColor	(*rgbin*)	(*rgbout*)
_MakeITable	(*colortab* INVERSETAB *res*)	
_ProtectEntry	(*index protect*)	
_QdError	()	(*qderr*)
_RealColor	(*color*)	(*realp*)
_ReserveEntry	(*index reserve*)	
_RestoreEntries	(*srctable dsttable selection*)	(*selectionout*)

_SaveEntries	(*srctable resulttable selection*)	(*selectionout*)
_SetClientID	(*id*)	
_SetEntries	(*start count atable*)	

Color-Picker

Lisp Function	*Arguments*	*Values Returned*
_CMY2RGB	(*ccolor rcolor*)	(*rcolorout*)
_Fix2SmallFract	(*f*)	(*s*)
_GetColor	(*where prompt incolor outcolor*)	(*normalexitp colorout*)
_HSL2RGB	(*hcolor rcolor*)	(*rcolorout*)
_HSV2RGB	(*hcolor rcolor*)	(*rcolorout*)
_RGB2CMY	(*rcolor ccolor*)	(*ccolorout*)
_RGB2HSL	(*rcolor hcolor*)	(*hcolorout*)
_RGB2HSV	(*rcolor hcolor*)	(*hcolorout*)
_SmallFract2Fix	(*s*)	(*f*)

Color-Quickdraw

Lisp Function	*Arguments*	*Values Returned*
_AllocCursor	()	
_BackPixPat	(*ppat*)	
_CalcCMask	(*srcbits dstbits srcrect dstrect seedrgb matchproc matchdata*)	
_CharExtra	(*extra*)	
_CloseCPort	(*port*)	
_CopyPixMap	(*srcpm dstpm*)	
_CopyPixPat	(*srcpp dstpp*)	
_DisposCCursor	(*crsrhndle*)	
_DisposCIcon	(*theicon*)	
_DisposCTable	(*ctable*)	
_DisposPixMap	(*pm*)	
_DisposPixPat	(*ppat*)	
_FillCArc	(*r startangle arcangle ppat*)	
_FillCOval	(*r ppat*)	
_FillCPoly	(*poly ppat*)	
_FillCRect	(*r ppat*)	
_FillCRgn	(*rgn ppat*)	
_FillCRoundRect	(*r ovwd ovht ppat*)	

_GetBackColor	(*color*)	(*colorout*)
_GetCCursor	(*crsrid*)	(*crsrhndle*)
_GetCIcon	(*id*)	(*theicon*)
_GetCPixel	(*h v cpix*)	(*cpixout*)
_GetCTable	(*ctid*)	(*ctable*)
_GetForeColor	(*color*)	(*colorout*)
_GetPixPat	(*patid*)	(*ppathandle*)
_HiliteColor	(*color*)	
_InitCport	(*port*)	
_MakeRGBPat	(*ppat mycolor*)	
_NewPixMap	()	(*pixmaphandle*)
_NewPixPat	()	(*ppat*)
_OpColor	(*color*)	
_OpenCport	(*port*)	
_PenPixPat	(*ppat*)	
_PlotIcon	(*therect theicon*)	
_RGBBackColor	(*color*)	
_RGBForeColor	(*color*)	
_SeedCFill	(*srcbits dstbits srcrect dstrect seedh seedv matchproc matchdata*)	
_SetCCursor	(*crsrhndle*)	
_SetCPixel	(*h v cpix*)	
_SetStdCProcs	(*cprocs*)	(*newcprocs*)

Color-Toolbox

Lisp Function	*Arguments*	*Values Returned*
_CWindowStructure	(*cwindow-pointer cwindow*)	(*out-window*)
_DelMCEntries	(*menuid menuitem*)	
_DispMCInfo	(*menucolortable*)	
_GetAuxCtl	(*thecontrol*)	(*owncolortabp achndl*)
_GetAuxWin	(*thewindow*)	(*existsp colortable*)
_GetCVariant	(*thecontrol*)	(*variantnumber*)
_GetGrayRgn	()	(*rgn*)
_GetItemCmd	(*themene item*)	(*cmdchar*)
_GetMCEntry	(*menuid menuitem*)	(*mcentryptr*)
_GetMCInfo	()	(*mctable*)
_GetNewCWindow	(*windowid wstorage behind*)	(*cwindow*)
_GetWVariant	(*whichwindow*)	(*number*)
_InitProcMenu	(*mbresid*)	
_MenuChoice	()	(*choice*)
_NewCDialog	(*dstorage boundsrect title*	(*dialogptr*)

	visible procid behind goawayflag refcon items)	
_NewCWindow	(*wstorage boundsrect title visible procid behind goawayflag refcon*)	(*cwindow*)
_PopUpMenuSelect	(*themenu top left popupitem*)	(*choice*)
_SetCtlColor	(*thecontrol newcolortable*)	
_SetDeskCPat	(*deskpixpat*)	
_SetItemCmd	(*themene item cmdchar*)	
_SetMCEntries	(*numentries menucentries*)	
_SetMCInfo	(*menucolortable*)	
_SetWinColor	(*thewindow newcolortable*)	

Control-Manager

Lisp Function	*Arguments*	*Values Returned*
_DisposeControl	(*thecontrol*)	
_DragControl	(*thecontrol startpt limitrect sloprect axis*)	
_Draw1Control	(*thecontrol*)	
_DrawControls	(*thewindow*)	
_FindControl	(*thepoint thewindow*)	(*whichcontrol*)
_GetCRefCon	(*thecontrol*)	(*value*)
_GetCTitle	(*thecontrol string*)	(*out-title*)
_GetCtlAction	(*thecontrol*)	(*actionproc*)
_GetCtlMax	(*thecontrol*)	(*value*)
_GetCtlMin	(*thecontrol*)	(*value*)
_GetCtlValue	(*thecontrol*)	(*value*)
_GetNewControl	(*controlid thewindow*)	(*thecontrol*)
_HideControl	(*thecontrol*)	
_HiliteControl	(*thecontrol highlight-state*)	
_KillControls	(*thewindow*)	
_MoveControl	(*thecontrol h v*)	
_NewControl	(*thewindow boundsrect title visible value min max procid refcon*)	(*thecontrol*)
_SetCRefCon	(*thecontrol value*)	
_SetCTitle	(*thecontrol title*)	
_SetCtlAction	(*thecontrol actionproc*)	
_SetCtlMax	(*thecontrol thevalue*)	
_SetCtlMin	(*thecontrol thevalue*)	
_SetCtlValue	(*thecontrol thevalue*)	
_ShowControl	(*thecontrol*)	
_SizeControl	(*thecontrol w h*)	
_TestControl	(*thecontrol thepoint*)	(*part-code*)

_TrackControl	(*thecontrol startpt actionproc*)	(*part-code*)
_UpdtControl	(*thewindow updatergn*)	

Desk Manager

Lisp Function	*Arguments*	*Values Returned*
_CloseDeskAcc	(*refnum*)	
_OpenDeskAcc	(*theacc*)	(*driver-ref*)
_SystemClick	(*theevent thewindow*)	
_SystemEdit	(*editcmd*)	(*deskaccp*)
_SystemEvent	(*theevent*)	(*deskaccp*)
_SystemMenu	(*menuresult*)	
_SystemTask	()	

Device-Manager

Lisp Function	*Arguments*	*Values Returned*
_CloseDriver	(*refnum*)	
_Control	(*refnum cscode csparamptr*)	
_GetDCtlEntry	(*refnum*)	(*controlhandle*)
_KillIO	(*refnum*)	
_OpenDriver	(*name*)	(*refnum*)
_PBControl	(*completion vrefnum refnum code param*)	
_PBKillIO	(*completion refnum*)	(*param*)
_PBStatus	(*completion vrefnum refnum code*)	(*param*)
_Status	(*refnum cscode csparamptr*)	

Dialog-Manager

Lisp Function	*Arguments*	*Values Returned*
_Alert	(*alertid filterproc*)	(*itemhit*)
_CautionAlert	(*alertid filterproc*)	(*itemhit*)
_CloseDialog	(*thedialog*)	
_CouldAlert	(*alertid*)	
_CouldDialog	(*dialogid*)	
_DialogSelect	(*theevent*)	(*deventp thedialog itemhit*)

_DisposDialog	(*thedialog*)	
_DlgCopy	(*thedialog*)	
_DlgCut	(*thedialog*)	
_DlgDelete	(*thedialog*)	
_DlgPaste	(*thedialog*)	
_DrawDialog	(*thedialog*)	
_ErrorSound	(*soundproc*)	
_FindDItem	(*thedialog thept*)	(*itemno*)
_FreeAlert	(*alertid*)	
_FreeDialog	(*dialogid*)	
_GetAlrtStage	()	(*stage*)
_GetDItem	(*thedialog itemno box*)	(*itemtype itemhandle outbox*)
_GetIText	(*item*)	(*text*)
_GetNewDialog	(*dialogid dstorage behind*)	(*dialogptr*)
_HideDItem	(*thedialog itemno*)	
_InitDialogs	(*resumeproc*)	
_IsDialogEvent	(*theevent*)	(*deventp*)
_ModalDialog	(*filterproc*)	(*itemhit*)
_NewDialog	(*dstorage boundsrect title visible procid behind goawayflag refcon items*)	(*dialogptr*)
_NoteAlert	(*alertid filterproc*)	(*itemhit*)
_ParamText	(*param0 param1 param2 param3*)	
_ResetAlrtStage	()	
_SelItext	(*thedialog itemno startsel endsel*)	
_SetDAFont	(*fontnum*)	
_SetDItem	(*thedialog itemno itemtype itemhandle box*)	
_SetIText	(*item text*)	
_ShowDItem	(*thedialog itemno*)	
_StopAlert	(*alertid filterproc*)	(*itemhit*)
_UpdtDialog	(*thedialog updateregion*)	

Disk-Driver

Lisp Function	*Arguments*	*Values Returned*
_DiskEject	(*drvnum*)	
_DriveStatus	(*drvnum statusin*)	(*statusout*)
_SetTagBuffer	(*buffptr*)	

Disk-Initialization

Lisp Function	*Arguments*	*Values Returned*
_DIBadMount	(*where evtmessage*)	(*result*)
_DIFormat	(*drvnum*)	
_DILoad	()	
_DIUnload	()	
_DIVerify	(*drvnum*)	
_DIZero	(*drvnum volname*)	

File-Manager

Lisp Function	*Arguments*	*Values Returned*
_Allocate	(*refnum allocation*)	(*actual-allocation*)
_Create	(*filename vrefnum creator filetype*)	
_Eject	(*volname vrefnum*)	
_FInitQueue	()	
_FlushVol	(*volname vrefnum*)	
_FSClose	(*refnum*)	
_FSDelete	(*filename vrefnum*)	
_FSOpen	(*filename vrefnum*)	(*refnum*)
_FSRead	(*refnum count buffer*)	(*result-count buffer-out*)
_FSWrite	(*refnum count buffer*)	(*result-count*)
_GetDrvQHdr	()	(*qheader*)
_GetEOF	(*refnum*)	(*logeof*)
_GetFInfo	(*filename vrefnum fndr-info*)	(*outcoming-fndr-info*)
_GetFPos	(*refnum*)	(*filepos*)
_GetFSQHdr	()	(*qheader*)
_GetVCBQHdr	()	(*qheader*)
_GetVInfo	(*drvnum*)	(*volname vrefnum freebytes*)
_GetVol	()	(*volname vrefnum*)
_GetVRefNum	(*pathrefnum*)	(*vrefnum*)
_OpenRF	(*filename vrefnum*)	(*refnum*)
_PBAllocate	(*refnum count*)	(*actual-count*)
_PBAllocContig	(*refnum count*)	(*actual-count*)
_PBCatMove	(*name vrefnum newname newdirid dirid*)	
_PBClose	(*refnum*)	
_PBCloseWD	(*vrefnum*)	

_PBCreate	(*name vrefnum version-number*)	
_PBDelete	(*name vrefnum version-number*)	
_PBDirCreate	(*name vrefnum version-number dirid*)	(*new-name new-dirid*)
_PBEject	(*name vrefnum*)	
_PBFlushFile	(*refnum*)	
_PBFlushVol	(*name vrefnum*)	
_PBGetCatInfo	(*hfileinfo name vrefnum fdirindex dirid*)	(*result-hfileinfo out-name*)
_PBGetEOF	(*refnum*)	(*eof*)
_PBGetFCBInfo	(*fcbpbrec name vrefnum refnum fcb-index*)	(*result-fcbpbrec out-name*)
_PBGetFInfo	(*fileparam name vrefnum version-number fdirindex*)	(*result-fileparam out-name*)
_PBGetFPos	(*refnum*)	(*mark*)
_PBGetVInfo	(*volumeparam name vrefnum volindex*)	(*result-volumeparam out-name*)
_PBGetVol	(*name*)	(*out-name vrefnum*)
_PBGetWDInfo	(*name vrefnum wdindex wdprocid wdvrefnum*)	(*out-name out-vrefnum out-wdprocid out-wdvrefnum out-wddirid*)
_PBHCreate	(*name vrefnum dirid*)	
_PBHDelete	(*name vrefnum dirid*)	
_PBHGetFInfo	(*hfileparam name vrefnum fdirindex dirid*)	(*result-hfileparam out-name*)
_PBHGetVInfo	(*hvolumeparam name vrefnum volindex*)	(*result-hvolumeparam out-name*)
_PBHGetVol	(*wdpbrecname*)	(*result-wdpbrec out-name*)
_PBHOpen	(*name vrefnum permission dirid*)	(*refnum*)
_PBHOpenRF	(*name vrefnum permission dirid*)	(*refnum*)
_PBHRstFLock	(*name vrefnum dirid*)	
_PBHSetFInfo	(*name vrefnum finder-info dirid creation-date modification-date*)	
_PBHSetFLock	(*name vrefnum dirid*)	
_PBHSetVol	(*name vrefnum wddirid*)	
_PBLockRange	(*refnum reqcount posmode posoffset*)	
_PBMountVol	(*vrefnum*)	(*result-vrefnum*)
_PBOffline	(*name vrefnum*)	
_PBOpen	(*name vrefnum versnum permission*)	(*refnum*)
_PBOpenRF	(*name vrefnum versnum permission*)	(*refnum*)
_PBOpenWD	(*name vrefnum wdprocid wddirid*)	(*new-wdrefnum*)
_PBRead	(*vrefnum refnum buffer count*	(*actual-count*

	posmode posoffset)	*out-buffer final-mark*)
_PBRename	(*name vrefnum version-number new-name*)	
_PBRstFLock	(*name vrefnum version-number*)	
_PBSetCatInfo	(*in-pb name*)	(*out-name*)
_PBSetEOF	(*refnum eof*)	
_PBSetFInfo	(*name vrefnum version-number finder-info creation-date modification-date*)	
_PBSetFLock	(*name vrefnum version-number*)	
_PBSetFPos	(*refnum posmode posoffset*)	(*new-mark*)
_PBSetFVers	(*name vrefnum version-number new-version-number*)	
_PBSetVInfo	(*name vrefnum creation-date modification-date volume-attributes clump-size backup-date sequence-number finder-info*)	
_PBSetVol	(*name vrefnum*)	
_PBUnlockRange	(*refnum reqcount posmode posoffset*)	
_PBUnmountVol	(*name vrefnum*)	
_PBWrite	(*vrefnum refnum buffer count posmode posoffset*)	(*actual-count new-mark*)
_Rename	(*oldname vrefnum newname*)	
_RstFLock	(*filename vrefnum*)	
_SetEOF	(*refnum logeof*)	
_SetFInfo	(*filename vrefnum fndr-info*)	
_SetFLock	(*filename vrefnum*)	
_SetFPos	(*refnum posmode posoff*)	
_SetVol	(*volname vrefnum*)	
_UnmountVol	(*volname vrefnum*)	

Font-Manager

Lisp Function	*Arguments*	*Values Returned*
_FMSwapFont	(*inrec*)	(*outrecptr*)
_FontMetrics	(*themetrics*)	(*thenewmetrics*)
_GetFNum	(*fontname*)	(*thenum*)
_GetFontName	(*fontnum*)	(*name*)
_InitFonts	()	
_RealFont	(*fontnum size*)	(*realp*)
_SetFontLock	(*lockp*)	
_SetFractEnable	(*fractp*)	
_SetFScaleDisable	(*disablep*)	

Graphics-Devices

Lisp Function	*Arguments*	*Values Returned*
_DisposGDevice	(*gdhandle*)	
_GetDeviceList	()	(*gdhandle*)
_GetGDevice	()	(*gdhandle*)
_GetMainDevice	()	(*gdhandle*)
_GetMaxDevice	(*globalrect*)	(*gdhandle*)
_GetNextDevice	(*gdh*)	(*gdhandle*)
_InitGDevice	(*gdrefnum mode gdh*)	
_NewGDevice	(*refnum mode*)	(*gdhandle*)
_SetDeviceAttribute	(*gdh attribute value*)	
_SetGDevice	(*gdh*)	
_TestDeviceAttribute	(*curdevice attribute*)	(*value*)

International-Utilities

Lisp Function	*Arguments*	*Values Returned*
_IUCompString	(*astr bstr*)	(*result*)
_IUDatePString	(*datetime form intlparam*)	(*result*)
_IUDateString	(*datetime form*)	(*result*)
_IUEqualString	(*astr bstr*)	(*result*)
_IUGetIntl	(*theid*)	(*irhandle*)
_IUMagIDString	(*aptr bptr alen blen*)	(*result*)
_IUMagString	(*aptr bptr alen blen*)	(*result*)
_IUMetric	()	(*metricp*)
_IUSetIntl	(*refnum theid intlparam*)	
_IUTimePString	(*datetime wantseconds intlparam*)	(*result*)
_IUTimeString	(*datetime wantseconds*)	(*result*)

List Manager

Lisp Function	*Arguments*	*Values Returned*
_LActivate	(*act lhandle*)	
_LAddColumn	(*count column lhandle*)	(*firstcolumnno*)
_LAddRow	(*count rownum lhandle*)	(*firstrownum*)

_LAddToCell	(*dataptr datalen thecell lhandle*)	
_LAutoScroll	(*lhandle*)	
_LCellSize	(*csize lhandle*)	
_LClick	(*pt modifiers lhandle*)	(*doubleclickp*)
_LClrCell	(*thecell lhandle*)	
_LDelColumn	(*count column lhandle*)	
_LDelRow	(*count rownum lhandle*)	
_LDispose	(*lhandle*)	
_LDoDraw	(*drawit lhandle*)	
_LDraw	(*thecell lhandle*)	
_LFind	(*thecell lhandle*)	(*offset len*)
_LGetCell	(*dataptr datalen thecell lhandle*)	
_LGetSelect	(*next thecellin lhandle*)	(*selectedp thecellout*)
_LLastClick	(*lhandle*)	(*lastcell*)
_LNew	(*rview databounds csize theproc thewindow drawit hasgrow scrollhoriz scrollvert*)	(*listhandle*)
_LNextCell	(*hnext vnext thecellin lhandle*)	(*cellsleftp thecellout*)
_LRect	(*cellrectin thecell lhandle*)	(*cellrectout*)
_LScroll	(*dcols drows lhandle*)	
_LSearch	(*dataptr datalen searchproc thecellin lhandle*)	(*foundp thecellout*)
_LSetCell	(*dataptr datalen thecell lhandle*)	
_LSetSelect	(*setit thecell lhandle*)	
_LSize	(*listwidth listheight lhandle*)	
_LUpdate	(*thergn lhandle*)	

Macintalk Manager

Lisp Function	*Arguments*	*Values Returned*
_MacinTalk	*(thespeech phonemes)*	
_Reader	*thespeech english phoneticoutput macintalk_refnum*)	
_SpeechOff	*(thespeech)*	
_SpeechOn	*(exceptions-file)*	*(thespeech file-refnum)*
_SpeechPitch	*(thespeech thepitch themode)*	
_SpeechRate	*(thespeech therate)*	

Memory Manager

Lisp Function	*Arguments*	*Values Returned*
_ApplicZone	()	(*appliczoneptr*)
_BlockMove	(*sourceptr destptr bytecount*)	
_CompactMem	(*cbneeded*)	(*cbytesavail*)
_DisposHandle	(*h*)	
_DisposPtr	(*p*)	
_EmptyHandle	(*h*)	
_FreeMem	()	(*freespace*)
_GetApplLimit	()	(*myptr*)
_GetHandleSize	(*h*)	(*handlesize*)
_GetPtrSize	(*p*)	(*ptrsize*)
_GetZone	()	(*currentzonethz*)
_GZSaveHnd	()	(*protectedhandle*)
_HandleZone	(*h*)	(*zoneptr*)
_HClrRBit	(*h*)	
_HGetState	(*h*)	(*state*)
_HLock	(*h*)	
_HNoPurge	(*h*)	
_HPurge	(*h*)	
_HSetRBit	(*h*)	
_HSetState	(*h state*)	
_HUnlock	(*h*)	
_InitApplZone	()	
_InitZone	(*pgrowzone cmoremasters limitptr startptr*)	
_MaxApplZone	()	
_MaxBlock	()	(*maxcb*)
_MaxMem	()	(*bytesavail zonegrowth*)
_MemError	()	(*err*)
_MoreMasters	()	
_MoveHHi	(*H*)	
_NewEmptyHandle	()	(*newhandle*)
_NewHandle	(*logicalsize*)	(*newhandle*)
_NewPtr	(*logicalsize*)	(*newptr*)
_PtrFromHandle	(*h*)	(*pntr*)
_PtrZone	(*p*)	(*zoneptr*)
_PurgeMem	(*cbneeded*)	
_PurgeSpace	()	(*total contig*)
_Read-Opaque-Bytes-From-Handle	(*h nbytes buffer*)	(*buffer-out*)
_Read-Opaque-Bytes-From-Pointer	(*ptr nbytes buffer*)	(*buffer-out*)
_ReallocHandle	(*h logicalsize*)	

_RecoverHandle	(*p*)	(*handle*)
_ResrvMem	(*cbneeded*)	
_SetApplBase	(*startptr*)	
_SetApplLimit	(*zonelimit*)	
_SetGrowZone	(*growzone*)	
_SetHandleSize	(*h newsize*)	
_SetPtrSize	(*p newsize*)	
_SetZone	(*hz*)	
_StackSpace	()	(*space*)
_SystemZone	()	(*systemzoneptr*)
_TopMem	()	(*topmem*)
_Write-Opaque-Bytes-Into-Handle-	(*h nbytes buffer*)	
_Write-Opaque-Bytes-Into-Pointer	(*ptr nbytes buffer*)	

Menu Manager

Lisp Function	*Arguments*	*Values Returned*
_AddResMenu	(*themenu thetype*)	
_AppendMenu	(*themenu data*)	
_CalcMenuSize	(*themenu*)	
_CheckItem	(*themenu item checked*)	
_ClearMenuBar	()	
_CountMItems	(*themenu*)	(*result*)
_DeleteMenu	(*menuid*)	
_DelMenuItem	(*themenu item*)	
_DisableItem	(*themenu item*)	
_DisposeMenu	(*themenu*)	
_DrawMenuBar	()	
_EnableItem	(*themenu item*)	
_FlashMenuBar	(*menuid*)	
_GetItem	(*themenu item itemstring*)	(*result-itemstring*)
_GetItemIcon	(*themenu item*)	(*result*)
_GetItemMark	(*themenu item*)	(*result*)
_GetItemStyle	(*themenu item*)	(*result*)
_GetMenu	(*resourceid*)	(*menu*)
_GetMenuBar	()	(*mbar*)
_GetMHandle	(*menuid*)	(*result*)
_GetNewMBar	(*menubarid*)	(*mbar*)
_HiliteMenu	(*menuid*)	
_InitMenus	()	
_*insertmenu*	(*themenu beforeid*)	
_*insertresmenu*	(*themenu thetype afteritem*)	
_InsMenuItem	(*themenu itemstring afteritem*)	
_MenuKey	(*ch*)	(*menu&item*)

_MenuSelect	(*startpt*)	(*menu&item*)
_NewMenu	(*menuid menutitle*)	(*menu*)
_SetItem	(*themenu item itemstring*)	
_SetItemIcon	(*themenu item icon*)	
_SetItemMark	(*themenu item markchar*)	
_SetItemStyle	(*themenu item chstyle*)	
_SetMenuBar	(*menulist*)	
_SetMenuFlash	(*count*)	

OS-Notification Manager

Lisp function	*Arguments*	*Value Returned*
_NMInstall	(*nmReqPtr QElemPtr*)	*OSErr*
_NMRemove	(*nmReqPtr QelemPtr*)	*OsErr*

OS-Event Manager

Lisp Function	*Arguments*	*Values Returned*
_FlushEvents	(*eventmask stopmask*)	
_GetEvQHdr	()	(*queue-header-ptr*)
_GetOSEvent	(*eventmask theeventin*)	(*eventp theeventout*)
_OSEventAvail	(*eventmask theeventin*)	(*eventp theeventout*)
_PostEvent	(*eventcode eventmsg*)	
_PPostEvent	(*eventcode eventmsg*)	(*qelptr*)
_SetEventMask	(*eventmask*)	

OS-Utilities Manager

Lisp Function	*Arguments*	*Values Returned*
_Date2Secs	(*date*)	(*secs*)
_Delay	(*numticks*)	(*finalticks*)
_Dequeue	(*qentry thequeue*)	
_Enqueue	(*qentry thequeue*)	
_Environs	()	(*rom machine*)
_EqualString	(*astr bstr casesens diacsens*)	(*equalp*)
_GetDateTime	()	(*secs*)
_GetMMUMode	()	(*mode*)
_GetSysPPtr	()	(*syspptr*)
_GetTime	(*datein*)	(*dateout*)
_GetTrapAddress	(*trapnum*)	(*trapaddr*)
_HandAndHand	(*ahndl bhndl*)	
_HandToHand	(*handlein*)	(*handleout*)
_InitUtil	()	

_NgetTrapAddress	(*trapnum traptype*)	(*trapaddr*)
_NSetTrapAddress	(*trapaddr trapnum traptype*)	
_PtrAndHand	(*pntr hndl size*)	
_PtrToHand	(*srcptr size*)	(*newhandle*)
_PtrToXHand	(*srcptr dsthndl size*)	
_ReadDateTime	()	(*secs*)
_RelString	(*astr bstr casesens diacsens*)	(*answer*)
_Restart	()	
_Restorea5	()	
_Secs2Date	(*secs datein*)	(*dateout*)
_SetDateTime	(*secs*)	
_SetTime	(*date*)	
_SetTrapAddress	(*trapaddr trapnum*)	
_SetUpA5	()	
_StripAddress	(*theaddress*)	(*strippedaddr*)
_SwapMMUMode	(*newmode*)	(*oldmode*)
_SysBeep	(*duration*)	
_UprString	(*stringin diacsens*)	(*stringout*)
_WriteParam	()	

Palette Manager

Lisp Function	*Arguments*	*Values Returned*
_ActivatePalette	(*srcwindow*)	
_AnimateEntry	(*dstwindow dstentry srcrgb*)	
_AnimatePalette	(*dstwindow srcctab srcindex dstentry dstlength*)	
_CTab2Palette	(*srcctab dstpalette srcusage srctolerance*)	
_DisposePalette	(*srcpalette*)	
_GetEntryColor	(*srcpalette srcentry dstrgbin*)	(*rgbout*)
_GetEntryUsage	(*srcpalette srcentry*)	(*dstusage dsttolerance*)
_GetNewPalette	(*paletteid*)	(*palettehndl*)
_GetPalette	(*srcwindow*)	(*palettehndl*)
_InitPalettes	()	
_NewPalette	(*entries srccolors srcusage srctolerance*)	(*palettehndl*)
_Palette2CTab	(*srcpalette destctab*)	
_PmBackColor	(*dstentry*)	
_PmForeColor	(*dstentry*)	
_SetEntryColor	(*dstpalette dstentry srcrgbin*)	
_SetEntryUsage	(*dstpalette dstentry srcusage srctolerance*)	
_SetPalette	(*dstwindow srcpalette*	

cupdates)

Printing Manager

Lisp Function	*Arguments*	*Values Returned*
_PrClose	()	
_PrCloseDoc	(*pprport*)	
_PrClosePage	(*pprport*)	
_PrCtlCall	(*iwhichctl lparam1 lparam2 lparam3*)	
_PrDrvrClose	()	
_PrDrvrDce	()	
_PrDrvrOpen	()	
_PrDrvrVers	()	(*version*)
_PrError	()	(*error*)
_PrGeneral	(*pdata*)	
_PrintDefault	(*hprint*)	
_PrJobDialog	(*hprint*)	(*comfirmedp*)
_PrJobMerge	(*hprintsrc hprintdst*)	
_PrOpen	()	
_PrOpenDoc	(*hprint pprport piobuf*)	(*theport*)
_PrOpenPage	(*pprport ppageframe*)	
_PrPicFile	(*hprint pprport piobuf pdevbuf prstatus*)	(*newprstatus*)
_PrSetError	(*err*)	
_PrStlDialog	(*hprint*)	(*comfirmedp*)
_PrValidate	(*hprint*)	(*notvalidp*)

QuickDraw Manager

Lisp Function	*Arguments*	*Values Returned*
_AddPt	(*srcpt1 srcpt2*)	(*outpt*)
_BackColor	(*color*)	
_BackPat	(*pat*)	
_CalcMask	(*srcptr dstptr srcrow dstrow height words*)	
_CharWidth	(*ch*)	(*width*)
_ClipRect	(*r*)	
_ClosePicture	()	
_ClosePoly	()	
_ClosePort	(*port*)	
_CloseRgn	(*dstrgn*)	

_ColorBit	(*whichbit*)	
_CopyBits	(*srcbits dstbits srcrect dstrect mode maskrgn*)	
_CopyMask	(*srcbits maskbits dstbits srcrect maskrect dstrect*)	
_CopyRgn	(*srcrgn dstrgn*)	
_DiffRgn	(*srcrgna srcrgnb dstrgn*)	
_DisposeRgn	(*rgn*)	
_DrawChar	(*ch*)	
_DrawPicture	(*mypicture dstrect*)	
_DrawString	(*s*)	
_DrawText	(*textbuf firstbyte bytecount*)	
_EmptyRect	(*r*)	(*emptyp*)
_EmptyRgn	(*rgn*)	(*emptyp*)
_EqualPt	(*pt1 pt2*)	(*equalp*)
_EqualRect	(*rect1 rect2*)	(*equalp*)
_EqualRgn	(*rgna rgnb*)	(*equalp*)
_EraseArc	(*r startangle arcangle*)	
_EraseOval	(*r*)	
_ErasePoly	(*poly*)	
_EraseRect	(*r*)	
_EraseRgn	(*rgn*)	
_EraseRoundRect	(*r ovalwidth ovalheight*)	
_FillArc	(*r startangle arcangle pat*)	
_FillOval	(*r pat*)	
_FillPoly	(*poly pat*)	
_FillRect	(*r pat*)	
_FillRgn	(*rgn pat*)	
_FillRoundRect	(*r ovalwidth ovalheight pat*)	
_ForeColor	(*color*)	
_FrameArc	(*r startangle arcangle*)	
_FrameOval	(*r*)	
_FramePoly	(*poly*)	
_FrameRect	(*r*)	
_FrameRgn	(*rgn*)	
_FrameRoundRect	(*r ovalwidth ovalheight*)	
_GetClip	(*rgn*)	
_GetFontInfo	(*info*)	(*out-info*)
_GetPen	()	(*outpt*)
_GetPenState	(*pnstate*)	(*outpnstate*)
_GetPixel	(*h v*)	(*onp*)
_GetPort	()	(*outport*)
_GlobalToLocal	(*pt*)	(*localpt*)
_GrafDevice	(*device*)	
_HideCursor	()	
_HidePen	()	
_InitCursor	()	
_InitGraf	()	

_InitPort	(*port*)	
_InsetRect	(*r dh dv*)	(*outrect*)
_InsetRgn	(*rgn dh dv*)	
_InvertArc	(*r startangle arcangle*)	
_InvertOval	(*r*)	
_InvertPoly	(*poly*)	
_InvertRect	(*r*)	
_InvertRgn	(*rgn*)	
_InvertRoundRect	(*r ovalwidth ovalheight*)	
_KillPicture	(*mypicture*)	
_KillPoly	(*poly*)	
_Line	(*dh dv*)	
_LineTo	(*h v*)	
_LocalToGlobal	(*pt*)	(*globalpt*)
_MapPoly	(*poly srcrect dstrect*)	
_MapPt	(*pt srcrect dstrect*)	(*outpt*)
_MapRect	(*r srcrect dstrect*)	(*newrect*)
_MapRgn	(*rgn srcrect dstrect*)	
_MeasureText	(*count textaddr charlocs*)	
_Move	(*dh dv*)	
_MovePortTo	(*leftglobal topglobal*)	
_MoveTo	(*h v*)	
_NewRgn	()	(*rgn*)
_ObscureCursor	()	
_OffsetPoly	(*poly dh dv*)	
_OffsetRect	(*r dh dv*)	(*outrect*)
_OffsetRgn	(*rgn dh dv*)	
_OpenPicture	(*picframe*)	(*pichandle*)
_OpenPoly	()	(*handle*)
_OpenPort	(*port*)	
_OpenRgn	()	
_PaintArc	(*r startangle arcangle*)	
_PaintOval	(*r*)	
_PaintPoly	(*poly*)	
_PaintRect	(*r*)	
_PaintRgn	(*rgn*)	
_PaintRoundRect	(*r ovalwidth ovalheight*)	
_PenMode	(*mode*)	
_PenNormal	()	
_PenPat	(*pat*)	
_PenSize	(*width height*)	
_PicComment	(*kind datasize datahandle*)	
_PortSize	(*width height*)	
_Pt2Rect	(*pt1 pt2 dstrect*)	(*outrect*)
_PtInRect	(*pt r*)	(*in_rect_p*)
_PtInRgn	(*pt rgn*)	(*inrgnp*)
_PtToAngle	(*r* PT)	(*angle*)
_Random	()	(*random*)

_RectInRgn	(*r rgn*)	(*inrgnp*)
_RectRgn	(*rgn* R)	
_ScalePt	(*pt srcrect dstrect*)	(*outpt*)
_ScrollRect	(*r dh dv updatergn*)	
_SectRect	(*src1 src2 dstrect*)	(*intersect-p outrect*)
_SectRgn	(*srcrgna srcrgnb dstrgn*)	
_SeedFill	(*srcptr dstptr srcrow dstrow height words seedh seedv*)	
_SetClip	(*rgn*)	
_SetCursor	(*crsrptr*)	
_SetCursorFromHandle		(*crsrhandle*)
_SetEmptyRgn	(*rgn*)	
_SetOrigin	(*h v*)	
_SetPenState	(*pnstate*)	
_SetPort	(*port*)	
_SetPortBits	(*bm*)	
_SetPt	(*h v*)	(*outpt*)
_SetRect	(*r left top right bottom*)	(*outrect*)
_SetRectRgn	(*rgn left top right bottom*)	
_SetStdProcs	(*procs*)	(*outprocs*)
_ShowCursor	()	
_ShowPen	()	
_SpaceExtra	(*extra*)	
_StdArc	(*verb r startangle arcangle*)	
_StdBits	(*srcbits srcrect dstrect mode maskrgn*)	(*newsrcbits newsrcrect newdstrect*)
_StdComment	(*kind datasize datahandle*)	
_StdGetPic	(*dataptr bytecount*)	
_StdLine	(*newpt*)	
_StdOval	(*verb r*)	
_StdPoly	(*verb poly*)	
_StdPutPic	(*dataptr bytecount*)	
_StdRect	(*verb r*)	
_StdRgn	(*verb rgn*)	
_StdRRect	(*verb r ovalwidth ovalheight*)	
_StdText	(*bytecount textbuf numer denom*)	
_StdTxMeas	(*bytecount textaddr numer denom info*)	(*width newnumer newdenom newfontinfo*)
_StringWidth	(*s*)	(*width*)
_StuffHex	(*thingptr s*)	
_SubPt	(*srcpt1 srcpt2*)	(*outpt*)
_TextFace	(*face*)	
_TextFont	(*font*)	
_TextMode	(*mode*)	
_TextSize	(*size*)	
_TextWidth	(*textbuf firstbyte*	(*width*)

	bytecount)	
_UnionRect	(*src1 src2 dstrect*)	(*outrect*)
_UnionRgn	(*srcrgna srcrgnb dstrgn*)	
_XorRgn	(*srcrgna srcrgnb dstrgn*)	

Resource Manager

Lisp Function	*Arguments*	*Values Returned*
_AddResource	(*thedata thetype theid name*)	
_ChangedResource	(*theresource*)	
_CloseResFile	(*refnum*)	
_Count1Resources	(*thetype*)	(*result*)
_Count1Types	()	(*result*)
_CountResources	(*thetype*)	(*result*)
_CountTypes	()	(*result*)
_CreateResFile	(*name*)	
_CurrentResLoad	()	(*current*)
_CurResFile	()	(*result*)
_DetachResource	(*theresource*)	
_Get1IndResource	(*thetype index*)	(*result*)
_Get1IndType	(*index*)	(*result*)
_Get1NamedResource	(*thetype name*)	(*result*)
_Get1Resource	(*thetype theid*)	(*result*)
_GetIndResource	(*thetype index*)	(*result*)
_GetIndType	(*index*)	(*result*)
_GetNamedResource	(*thetype name*)	(*result*)
_GetResAttrs	(*theresource*)	(*result*)
_GetResFileAttrs	(*refnum*)	(*result*)
_GetResInfo	(*theresource name*)	(*theid thetype out-name*)
_GetResource	(*thetype theid*)	(*result*)
_HomeResFile	(*theresource*)	(*result*)
_InitResources	()	(*result*)
_LoadResource	(*theresource*)	
_MaxSizeRsrc	(*theresource*)	(*result*)
_OpenResFile	(*name*)	(*result*)
_OpenRFPerm	(*filename vrefnum permission*)	(*result*)
_ReleaseResource	(*theresource*)	
_ResError	()	(*result*)
_RGetResource	(*thetype theid*)	(*rhandle*)
_RmveResource	(*theresource*)	
_RsrcMapEntry	(*theresource*)	(*result*)
_RsrcZoneInit	()	
_SetResAttrs	(*theresource attrs*)	
_SetResFileAttrs	(*refnum attrs*)	

_SetResInfo	(*theresource theid name*)	
_SetResLoad	(*load*)	
_SetResPurge	(*install*)	
_SizeResource	(*theresource*)	(*result*)
_Unique1ID	(*thetype*)	(*result*)
_UniqueID	(*thetype*)	(*result*)
_UpdateResFile	(*refnum*)	
_UseResFile	(*refnum*)	
_WriteResource	(*theresource*)	

Scrap-Resource Manager

Lisp Function	*Arguments*	*Values Returned*
_GetScrap	(*hdest thetype*)	(*offset*)
_GetScrapStuff	(*scrapstuff*)	(*newstuff*)
_InfoScrap	()	(*scrapstuffptr*)
_LoadScrap	()	
_PutScrap	(*length thetype source*)	
_UnloadScrap	()	
_ZeroScrap	()	

Script Manager

Lisp Function	*Arguments*	*Values Returned*
_Char2Pixel	(*textbuf textlen slop offset direction*)	(*pixelwidth*)
_CharByte	(*textbuf textoffset*)	(*chartype*)
_CharType	(*textbuf textoffset*)	(*chartype*)
_DrawJust	(*textptr textlength slop*)	
_FindWord	(*textptr textlength offset leftside breaks offsetsin*)	(*offsetsout*)
_Font2Script	(*fontnumber*)	(*scriptcode*)
_FontScript	()	(*scriptcode*)
_GetAppFont	()	(*fontnum*)
_GetDefFontSize	()	(*size*)
_GetEnvirons	(*verb*)	(*param*)
_GetMBarHeight	()	(*height*)
_GetScript	(*script verb*)	(*param*)
_GetSysFont	()	(*fontnume*)
_GetSysJust	()	(*just*)
_HiliteText	(*textptr textlength firstoffset secondoffset offsetsin*)	(*offsetsout*)
_IntlScript	()	(*scriptcode*)
_KeyScript	(*scriptcode*)	
_MeasureJust	(*textptr textlength slop*	

	charlocs)	
_Pixel2Char	(*textbuf textlen slop pixelwidth*)	(*offset leftside-p*)
_SetEnvirons	(*verb param*)	
_SetScript	(*script verb param*)	
_SetSysJust	(*newjust*)	
_Transliterate	(*srchandle dsthandle target srcmask*)	(*result*)

SCSI Manager

Lisp Function	*Arguments*	*Values Returned*
_SCSICmd	(*buffer count*)	
_SCSIComplete	(*wait*)	(*stat message*)
_SCSIGet	()	
_SCSImsgin	()	(*message*)
_SCSIMsgOut	(*message*)	
_SCSIRBlind	(*tibptr*)	
_SCSIRead	(*tibptr*)	
_SCSIReset	()	
_SCSIselatn	(*targetid*)	
_SCSISelect	(*targetid*)	
_SCSIStat	()	(*bits*)
_SCSIWBlind	(*tibptr*)	
_SCSIWrite	(*tibptr*)	

Segment-Loader Manager

Lisp Function	*Arguments*	*Values Returned*
_ClrAppFiles	(*index*)	
_CountAppFiles	()	(*message count*)
_ExitToShell	()	
_GetAppFiles	(*index thefile*)	(*thefileout*)
_GetAppParms	()	(*name aprefnum apparam*)
_UnloadSeg	(*routineaddr*)	

Serial-Driver Manager

Lisp Function	*Arguments*	*Values Returned*
_RAMSDClose	(*whichport*)	
_RAMSDOpen	(*whichport*)	

_SerClrBrk	(*refnum*)	
_SerGetBuf	(*refnum*)	(*count*)
_SerHShake	(*refnum flags*)	
_SerReset	(*refnum serconfig*)	
_SerSetBrk	(*refnum*)	
_SerSetBuf	(*refnum serbptr serblen*)	
_SerStatus	(*refnum serstatin*)	(*serstatout*)

Slot Manager

Lisp Function	*Arguments*	*Values Returned*
_InitPRAMRecs	()	
_InitSDeclMgr	()	
_InitsRsrcTable	()	
_sCalcsPointer	(*spspointer spoffsetdata spbytelanes*)	
_sCalcStep	(*spspointer spbytelanes spflags*)	(*spresult*)
_sCardChanged	(*spslot*)	(*spresult*)
_sCkCardStatus	(*spslot*)	(*spresult*)
_sdeleteSRTRec	(*spslot spid spextdev*)	
_sExec	(*spspointer spid spsexecpblk*)	(*spresult*)
_sFindDevBase	(*spslot spid*)	(*spresult*)
_sFindsInfoRecPtr	(*spslot*)	(*spresult*)
_sFindsRsrcPtr	(*spslot spid*)	(*spspointer*)
_sFindStruct	(*spid spspointer*)	(*new-spspointer spbytelanes*)
_sGetBlock	(*spspointer spid*)	(*spresult spoffsetdata spbytelanes spsize spflags*)
_sGetcString	(*spspointer spid*)	(*spresult spoffsetdata spbytelanes spsize spflags*)
_sGetDriver	(*spslot spid spextdev spsexecpblk*)	(*spresult spflags spsize*)
_SNextsRsrc	(*spslot spid spextdev*)	(*new-spslot new-spid new-spextdev spspointer sprefnum spioreserved spcategory spctype spdrvrsw spdrvrhw sphwdev*)
_sNextTypesRsrc	(*spslot spid spextdev sptbmask spcategory spctype spdrvrsw spdrvrhw sphwdev*)	(*new-spslot new-spid new-spextdev spspointer sprefnum spioreserved new-spcategory new-spctype new-spdrvrsw new-spdrvrhw new-sphwdev*)

_sOffsetData	(*spspointer spid*)	(*spoffsetdata spbytelanes spresult spflags*)
_sPrimaryInit	(*spflags*)	
_sPtrToSlot	(*spspointer*)	(*spslot*)
_sPutPRAMRec	(*spslot spspointer*)	
_sReadByte	(*spspointer spid*)	(*spresult spoffsetdata spbytelanes*)
_sReadDrvrName	(*spslot spid spresult*)	(*spsize spspointer*)
_sReadFHeader	(*spslot spresult*)	(*spspointer spbytelanes spsize spoffsetdata*)
_sReadInfo	(*spslot spresult*)	(*spsize*)
_sReadLong	(*spspointer spid*)	(*spresult spoffsetdata spbytelanes spsize*)
_sReadPBSize	(*spspointer spid spflags*)	(*spsize spbytelanes spresult*)
_sReadPRAMRec	(*spslot spresult*)	(*spsize*)
_sReadStruct	(*spspointer spsize spresult*)	(*spbytelanes*)
_sReadWord	(*spspointer spid*)	(*spresult spoffsetdata spbytelanes*)
_sRsrcInfo	(*spslot spid spextdev*)	(*spspointer spioreserved sprefnum spcategory spctype spdrvrsw spdrvrhw sphwdev*)
_sSearchSRT	(*spslot spid spextdev spflags spspointer*)	
_sUpdateSRT	(*spslot spid spextdev sprefnum spioreserved*)	(*spspointer spflags spsize spresult*)

Sound Manager

Lisp Function	*Arguments*	*Values Returned*
_GetSoundVol	()	(*level*)
_SetSoundVol	(*level*)	
_SndAddModifier	(*chan modifier id init*)	
_SndControl	(*id cmd*)	(*cmdout*)
_SndDisposeChannel	(*chan quitnow*)	
_SndDoCommand	(*chan cmd nowait*)	
_SndDoImmediate	(*chan cmd*)	
_SndNewChannel	(*chan synth init userroutine*)	(*newchan*)
_SndPlay	(*channel sndhdl async*)	
_SoundDone	()	(*donep*)
_StartSound	(*synthrec numbytes completionrtn*)	
_StopSound	()	

Standard-File-Package Manager

Lisp Function	*Arguments*	*Values Returned*
_SFGetFile	(*where prompt filefilter numtypes typelist dialoghook reply*)	(*replyout*)
_SFPGetFile	(*where prompt filefilter numtypes typelist dialoghook reply dlgid filterproc*)	(*replyout*)
_SFPPutFile	(*where prompt origname dlghook reply dlgid filterproc*)	(*replyout*)
_SFPutFile	(*where prompt origname dlghook reply*)	(*replyout*)

System-Misc Manager

Lisp Function	*Arguments*	*Values Returned*
_DTInstall	(*dttaskptr*)	
_InitAllPacks	()	
_InitPack	(*packid*)	
_InsTime	(*tmtaskptr*)	
_NumToString	(*thenum*)	(*thestring*)
_PrimeTime	(*tmtaskptr count*)	
_RmvTime	(*tmtaskptr*)	
_ShutDwnInstall	(*shutdwnproc flags*)	
_ShutDwnPower	()	
_ShutDwnRemove	(*shutdwnproc*)	
_ShutDwnStart	()	
_StringToNum	(*thestring*)	(*thenum*)
_SysEnvirons	(*versionRequested*)	(*theWorld*)
_SysError	(*errorcode*)	

TextEdit Manager

Lisp Function	*Arguments*	*Values Returned*
_GetStylHandle	(*hte*)	(*stylehandle*)
_GetStylScrap	(*hte*)	
_SetClikLoop	(*clikproc hte*)	
_SetStylHandle	(*thehandle hte*)	

_SetWordBreak	(*wbrkproc hte*)	
_TEActivate	(*hte*)	
_TEAutoView	(*autoview hte*)	
_TECalText	(*hte*)	
_TEClick	(*pt extend hte*)	
_TECopy	(*hte*)	
_TECut	(*hte*)	
_TEDeactivate	(*hte*)	
_TEDelete	(*hte*)	
_TEDispose	(*hte*)	
_TEFromScrap	()	
_TEGetHeight	(*endline startline hte*)	(*height*)
_TEGetOffset	(*pt hte*)	(*offset*)
_TEGetPoint	(*offset hte*)	(*pt*)
_TEGetScrapLen	()	(*length*)
_TEGetStyle	(*offset thestyle hte*)	(*style lineheight fontascent*)
_TEGetText	(*hte*)	(*charshandle*)
_TEIdle	(*hte*)	
_TEInit	()	
_TEInsert	(*text length hte*)	
_TEKey	(*key hte*)	
_TENew	(*destrect viewrect*)	(*handle*)
_TEPaste	(*hte*)	
_TEPinScroll	(*dh dv hte*)	
_TEReplaceStyle	(*mode oldstyle newstyle redraw hte*)	
_TEScrapHandle	()	(*scraphandle*)
_TEScroll	(*dh dv hte*)	
_TESelView	(*hte*)	
_TESetJust	(*just hte*)	
_TESetScrapLen	(*length*)	
_TESetSelect	(*selstart selend hte*)	
_TESetStyle	(*mode newstyle redraw hte*)	
_TESetText	(*text length hte*)	
_TEStylInsert	(*text length hst hte*)	
_TEStylNew	(*destrect viewrect*)	(*handle*)
_TEStylPaste	(*hte*)	
_TEToScrap	()	
_TEUpdate	(*rupdate hte*)	
_TextBox	(*text length box just*)	

Toolbox-Event Manager

Lisp Function	*Arguments*	*Values Returned*
_Button	()	(*downp*)
_EventAvail	(*eventmask theevent*)	(*handle-event-p nextevent*)

_GetCaretTime	()	(*carettime*)
_GetDblTime	()	(*dbltime*)
_GetKeys	(*keymapin*)	(*keymapout*)
_GetMouse	()	(*mouseloc*)
_GetNextEvent	(*eventmask theevent*)	(*handle-event-p nextevent*)
_KeyTrans	(*transdata keycode*)	(*result state*)
_StillDown	()	(*stilldownp*)
_TickCount	()	(*tickcount*)
_WaitMouseUp	()	(*stilldownp*)

Toolbox-Utilities Manager

Lisp Function	*Arguments*	*Values Returned*
_AngleFromSlope	(*slope*)	(*angle*)
_BitAnd	(*value1 value2*)	(*result*)
_BitClr	(*byteptr bitnum*)	
_BitNot	(*value*)	(*result*)
_BitOr	(*value1 value2*)	(*result*)
_BitSet	(*byteptr bitnum*)	
_BitShift	(*value count*)	(*result*)
_BitTst	(*byteptr bitnum*)	(*resultp*)
_BitXor	(*value1 value2*)	(*result*)
_DeltaPoint	(*pta ptb*)	(*longresult*)
_FixMul	(*a b*)	(*answer*)
_FixRatio	(*numer denom*)	(*fixed*)
_FixRound	(*x*)	(*int*)
_GetCursor	(*cursorid*)	(*chandle*)
_GetIcon	(*iconid*)	(*iconhandle*)
_GetIndPattern	(*thepattern patlistid index*)	(*thepatternout*)
_GetIndString	(*stringin strlistid index*)	(*stringout*)
_GetPattern	(*patid*)	(*phandle*)
_GetPicture	(*picid*)	(*phandle*)
_GetString	(*stringid*)	(*strhandle*)
_HiWord	(*x*)	(*hiword*)
_LongMul	(*a b*)	(*answer*)
_LoWord	(*x*)	(*loword*)
_Munger	(*h offset ptr1 len1 ptr2 len2*)	(*answer*)
_NewString	(*thestring*)	(*strhandle*)
_PackBits	(*srcptrin dstptrin srcbytes*)	(*srcptrout dstptrout*)
_PlotIcon	(*therect theicon*)	
_ScreenRes	()	(*scrnhres screenvres*)
_SetString	(*h thestring*)	
_ShieldCursor	(*shieldrect offsetpt*)	
_SlopeFromAngle	(*angle*)	(*slope*)
_UnpackBits	(*srcptrin dstptrin srcbytes*)	(*srcptrout dstptrout*)

Vertical-Retrace Manager

Lisp Function	*Arguments*	*Values Returned*
_AttachVBL	(*theslot*)	
_DoVBLTask	(*theslot*)	
_GetVBLQHdr	()	(*qhdrptr*)
_SlotVInstall	(*vbltaskptr theslot*)	
_SlotVRemove	(*vbltaskptr theslot*)	
_VInstall	(*vbltaskptr*)	
_VRemove	(*vbltaskptr*)	

Window Manager

Lisp Function	*Arguments*	*Values Returned*
_BeginUpdate	(*thewindow*)	
_BringToFront	(*window*)	
_CalcVis	(*window*)	
_CalcVisBehind	(*startwindow clobberedrgn*)	
_CheckUpdate	(*theevent*)	(*new-event found-p*)
_ClipAbove	(*window*)	
_CloseWindow	(*thewindow*)	
_DisposeWindow	(*thewindow*)	
_DragGrayRgn	(*thergn startpt limitrect sloprect axis actionproc*)	(*point_difference*)
_DragWindow	(*thewindow startpt boundsrect*)	
_DrawGrowIcon	(*thewindow*)	
_DrawNew	(*window update*)	
_EndUpdate	(*thewindow*)	
_FindWindow	(*thepoint*)	(*whichwindow kind*)
_FrontWindow	()	(*window*)
_GetNewWindow	(*windowid behind*)	(*newwindow*)
_GetWindowPic	(*thewindow*)	(*pic*)
_GetWMgrPort	()	(*wport*)
_GetWRefCon	(*thewindow*)	(*data*)
_GetWTitle	(*thewindow in-title*)	(*out-title*)
_GrowWindow	(*thewindow startpt sizerect*)	(*portsize*)
_HideWindow	(*thewindow*)	
_HiliteWindow	(*thewindow fhilite*)	
_InitWindows	()	
_InvalRect	(*badrect*)	
_InvalRgn	(*badrgn*)	
_MoveWindow	(*thewindow hglobal vglobal front*)	
_NewWindow	(*boundsrect title visible procid behind goawayflag*	(*newwindow*)

	refcon)	
_PaintBehind	(*startwindow clobberedrgn*)	
_PaintOne	(*window clobberedrgn*)	
_PinRect	(*therect thept*)	(*nearest-point*)
_SaveOld	(*window*)	
_SelectWindow	(*thewindow*)	
_SendBehind	(*thewindow behindwindow*)	
_SetWindowPic	(*thewindow pic*)	
_SetWRefCon	(*thewindow data*)	
_SetWTitle	(*thewindow new-title*)	
_ShowHide	(*thewindow showflag*)	
_ShowWindow	(*thewindow*)	
_SizeWindow	(*thewindow width height fupdate*)	
_TrackBox	(*thewindow thepoint partcode*)	(*result*)
_TrackGoAway	(*thewindow thepoint*)	(*result*)
_ValidRect	(*goodrect*)	
_ValidRgn	(*goodrgn*)	
_WindowStructure	(*window-pointer window*)	(*out-window*)
_ZoomWindow	(*thewindow partcode front*)	

MacIvory Error Conditions

This appendix describes MacIvory error conditions. The Macintosh error name, the corresponding Lisp error condition and the error message are given for each error condition. Remote error flavors for the toolbox routines are in the **macintosh-internals** package.

The errors are listed in alphabetical order by Macintosh error name.

`abortErr` **mac-os-error-aborterr**
I/O request aborted by KillIO

`addResFailed` **mac-os-error-addresfailed**
AddResource failed

`badBtSlpErr` **mac-os-error-badbtslperr**
Bad address mark (btslp)

`badChannel` **mac-os-error-badchannel**
Invalid channel queue length

`badCksmErr` **mac-os-error-badcksmerr**
Bad address mark (cksum)

`badDBtSlp` **mac-os-error-baddbtslp**
Bad data mark (btslp)

`badDCksum` **mac-os-error-baddcksum**
Bad data mark (cksum)

`badFormat` **mac-os-error-badformat**
Handle to snd resource was invalid

`badMDBErr` **mac-os-error-badmdberr**
Bad master directory block; must reinitialize volume

`badMovErr` **mac-os-error-badmoverr**
Attempted to move into offspring

`badUnitErr` **mac-os-error-baduniterror**
Driver reference number doesn't match unit table

`bdNamErr` **mac-os-error-bdnamerr**
Bad file name or volume name (perhaps zero-length)

`cantStepErr` **mac-os-error-cantsteperr**
Drive error (step)

`clkRdErr` **mac-os-error-clkrderr**
Unable to read clock

`clkWrErr` **mac-os-error-clkwerr**
Time written did not verify

`controlErr` **mac-os-error-controlerr**
Driver can't respond to this Control call

`corErr` **mac-os-error-corerr**
Trap ("core routine") number out of range

`dataVerErr` **mac-os-error-dataververr**
Read-verify failed

`dInstErr` **mac-os-error-dinsterr**
Couldn't find driver in resource file

`dirFulErr` **mac-os-error-dirfulerr**
File directory full

`dirNFErr` **mac-os-error-dirnferr**
Directory not found

`dRemovErr` **mac-os-error-dremoverr**
Attempt to remove an open driver

`dskFulErr` **mac-os-error-dskfulerr**
All allocation blocks on the volume are full

`dupFNErr` **mac-os-error-dupfnerr**
File with specified name and version number already exists

`envNotPresent` **mac-os-error-envnotpresent**
Sysenvirons trap not present, system file earlier than version 4.1.

`eofErr` **mac-os-error-eoferr**
Logical end-of-file reached during read operation

`evtNotEnb` **mac-os-error-evtnotenb**
Event type not designated in system event mask

`extFSErr` **mac-os-error-extfserr**
External file system; file-system identifier is nonzero, or path reference number is >1024

`fBsyErr` **mac-os-error-fbsyerr**
File is busy, one or more files are open

`fLckdErr` **mac-os-error-flckderr**
File is locked

`fnfErr` **mac-os-error-fnferr**
File not found

`fnOpnErr` **mac-os-error-fnopnerr**
File not open

`fsDSIntErr` **mac-os-error-fsdsinterr**
Internal file system error

`fsRnErr` **mac-os-error-fsrnerr**
Problem during rename

`gfpErr` **mac-os-error-gfperr**
Error during GetFPos

`iIOAbort` **mac-os-error-iioabort**
I/O Error

`iMemFullErr` **mac-os-error-imemfullerr**
Not enough room in heap zone

`initIWMErr` **mac-os-error-initiwmerr**
Can't initialize disk controller chip

`ioErr` **mac-os-error-ioerr**
I/O error

`iPrAbort` **mac-os-error-iprabort**
Application or user requested abort

`iPrSavPFil` **mac-os-error-iprsavpfil**
Saving spool file

`memAZErr` **mac-os-error-memazerr**
Undocumented Memory Manager error -113

`memFullErr` **mac-os-error-memfullerr**
Not enough room in heap zone

`memLockedErr` **mac-os-error-memlockederr**
Block is locked

`memPurErr` **mac-os-error-mempurerr**
Attempt to purge a locked block

`memROZErr` **mac-os-error-memrozerr**
operation on read-only zone

`memWZErr` **mac-os-error-memwzerr**
Attempt to operate on a free block

`nilHandleErr` **mac-os-error-nilhandleerr**
NIL master pointer

`noAdrMkErr` **mac-os-error-noadrmkerr**
Can't find an address mark

`NoDriveErr` **mac-os-error-nodriveerr**
Drive isn't connected

`noDtaMkErr` **mac-os-error-nodtamkerr**
Can't find a data mark

`noHardware` **mac-os-error-nohardware**
No Hardware support for the specified synthesizer

`noMacDskErr` **mac-os-error-nomacdskerr**
Not a Macintosh disk; volume lacks Macintosh-format directory

`noNybErr` **mac-os-error-nonyberr**
Disk is probably blank

`noScrapErr` **mac-os-error-noscraperr**
Desk scrap isn't initialized

`notEnoughHardware` **mac-os-error-notenoughhardware**
No more channels for the specified synthesizer

`notOpenErr` **mac-os-error-notopenerr**
Driver isn't open

`noTypeErr` **mac-os-error-notypeerr**
No data of the requested type

`nsDrvErr` **mac-os-error-nsdrverr**
No such drive; specified drive number doesn't match any number in the drive queue

`nsvErr` **mac-os-error-nsverr**
Specified volume doesn't exist

`offLinErr` **mac-os-error-offlinerr**
No disk in drive

`openErr` **mac-os-error-openerr**
Requested read/write permission doesn't match driver's open permission

`opWrErr` **mac-os-error-opwrerr**
The read/write permission of only one access path to a file can allow writing

`paramErr` **mac-os-error-paramerr**
Error in parameter list

`permErr` **mac-os-error-permerr**
Attempt to open locked file for writing

`posErr` **mac-os-error-poserr**
Attempt to position before start of file

`prInitErr` **mac-os-error-priniterr**
Validity status is not $A8

`prWrErr` **mac-os-error-prwrerr**
Parameter RAM written did not verify

`qErr` **mac-os-error-qerr**
Entry not in queue

`queueFull` **mac-os-error-queuefull**
No room in queue

`readErr` **mac-os-error-readerr**
Driver can't respond to Read calls

`resFNotFound` **mac-os-error-resfnotfound**
Resource file not found

`resNotFound` **mac-os-error-resnotfound**
Resource not found

`resProblem` **mac-os-error-resproblem**
Problem loading resource

`rfNumErr` **mac-os-error-rfnumerr**
Path reference number specifies nonexistent access path

`rmvResFailed` **mac-os-error-rmvresfailed**
RmveResource failed

`sectNFErr` **mac-os-error-sectnferr**
Can't find sector

`seekErr` **mac-os-error-seekerr**
Drive error (seek)

`slotNumErr` **mac-os-error-slotnumerr**
Invalid slot number

`spdAdjErr` **mac-os-error-spdadjerr**
Can't correctly adjust disk speed

`statusErr` **mac-os-error-statuserr**
Driver can't respond to this Status call

`tk0BadErr` **mac-os-error-tk0baderr**
Can't find track 0

`tmfoErr` **mac-os-error-tmfoerr**
Too many files open

`tmwdoErr` **mac-os-error-tmwdoerr**
Too many working directories open

`twosideErr` **mac-os-error-twosideerr**
Tried to read side 2 of a disk in a single read

`unimpErr` **mac-os-error-unimperr**
Unimplemented trap

`unitEmptyErr` **mac-os-error-unitemptyerr**
Driver reference number specifies NIL handle in unit table

`vLckdErr` **mac-os-error-vlckderr**
Volume is locked

`volOffLinErr` **mac-os-error-volofflinerr**
Volume not on-line

`volOnLinErr` **mac-os-error-volonlinerr**
Specified volume is already mounted and on-line

`vTypErr` **mac-os-error-vtyperr**
QType field of entry in vertical retrace queue isn't vType

`wPrErr` **mac-os-error-wprerr**
Volume is locked by a hardware setting

`wrgVolTypErr` **mac-os-error-wrgvoltyperr**
Attempt to do hierarchical operation on nonhierarchical volume

`writErr` **mac-os-error-writerr**
Driver can't respond to Write calls

`wrPermErr` **mac-os-error-wrpermerr**
Read/write permission doesn't allow writing

`wrUnderrun` **mac-os-error-wrunderrun**
Write underrun occurred