## Greeting and Initialization Files

Many of the features of Medley are controlled by variables that you can adjust to your own taste. In addition, you can modify the action of system functions in ways not specifically provided for by using ADVISE (see the Advise Functions section of Chapter 15). To encourage customizing Medley's environment, it includes a facility for automatically loading initialization files (or "init files") when it is first started. Each user can have a separate "user init file" that customizes Medley's environment to his/her tastes. In addition, there can be a "site init file" that applies to all users at a given physical site, setting system variables that are the same for all users such as the name of the nearest printer, etc.

The process of loading init files, also known as "greeting", occurs when a Medley system created by MAKESYS (see the Saving Virtual Memory State section below) is started for the first time. The user can also explicitly invoke the greeting operation at any time via the function GREET (below). The process of greeting includes the following steps:

1. Any previous greeting operation is undone. The side effects of the greeting operation are stored on a global variable as well as on the history list, thus enabling the previous greeting to be undone even if it has dropped off of the bottom of the history list.
2. All of the items on the list PREGREETFORMS are evaluated.
3. The site init file is loaded. GREET looks for a file by the name {DSK}INIT.LISP. If this is found, it is loaded. If it is not found, the system prints `Please enter name of system init file (e.g. {server}<directory>INIT.extension):` and waits for you to type a file name, followed by a carriage return. If you just type a carriage return without typing a file name, no site init file is loaded. **Note**: The site init file is loaded with *LDFLG* set to SYSLOAD, so that no file package information is saved, and nothing is printed out.
4. The user init file is loaded. The user init file is found by using the variable USERGREETFILES (described below), which is normally set in the site init file. The user init file is loaded with normal file package settings, but under errorset protection and with PRETTYHEADER set to NIL to suppress the `File created` message.
5. All of the items on the list POSTGREETFORMS are evaluated.
6. The greeting "`Hello, `*XXX*`.`" is printed, where *XXX* is the value of the variable FIRSTNAME (if non-NIL). The variable GREETDATES (below) can be set to modify this greeting for particular dates.

(**GREET** *NAME* —) [Function]

Performs the greeting for person whose username is *NAME* (if *NAME* = NIL, uses the login name). When Medley first starts up, it performs (GREET).

(**GREETFILENAME** *USER*) [Function]

If *USER* is T, GREETFILENAME returns the file name of the site init file. If the file name doesn't exist, you are prompted for it. Otherwise, *USER* is interpreted to be a user's system name, and GREETFILENAME returns the file name for the user init file (if it exists).

**USERGREETFILES** [Variable]

USERGREETFILES specifies a series of file names to try as the user init file. The value of USERGREETFILES is a list, where each element is a list of symbols. For each item in USERGREETFILES, the user name is substituted for the symbol USER and the value of COMPILE.EXT (see the Cimpiler Functions section of Chapter 18) is substituted for the symbol COM, and the symbols are packed into a single file name. The first such file that is found is the user init file.

For example, suppose that the value of USERGREETFILES was

```
(({ERIS}< USER >LISP>INIT. COM)
 ({ERIS}< USER >LISP>INIT)
 ({ERIS}< USER >INIT. COM)
 ({ERIS}< USER >INIT))
```

If the user name was JONES, and the value of COMPILE.EXT was DCOM, then this would search for the files {ERIS}<JONES>LISP>INIT.DCOM, {ERIS}<JONES>LISP>INIT, {ERIS}<JONES>INIT.DCOM, and {ERIS}<JONES>INIT.

**Note**: The file name "specifications" in USERGREETFILES should be fully qualified, including all host and directory information. The directory search path (the value of DIRECTORIES, see the Searching File Directories section of Chapter 24) is *not* used to find the user greet files.

**GREETDATES** [Variable]

The value of GREETDATES can be used to specify special greeting messages for various dates. GREETDATES is a list of elements of the form (*DATESTRING* . *STRING*), e.g. ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her INIT.LISP file by using a ADDVARS file package command like (ADDVARS (GREETDATES ("8-FEB" . "Happy Birthday"))). On the specified date, the GREET will use the indicated salutation.

It is impossible to give a complete list of all of the variables and functions you may want to set in your init files. The default values for system variables are chosen in the hope that they will be correct for the majority of users, so many users get along with very small init files. The following describes some of the variables that users may want to reset in their init files:

Directories The variables DIRECTORIES and LISPUSERSDIRECTORIES (see the Searching File Directories section of Chapter 24) contain lists of directories used when searching for files. LOGINHOST/DIR (see the Incomplete File Names section of Chapter 24) determines the default directory used when you call CONN with no argument.

Fonts and Printing — The variables DISPLAYFONTDIRECTORIES, DISPLAYFONTEXTENSIONS, INTERPRESSFONTDIRECTORIES, and PRESSFONTWIDTHSFILES (see the Font Files and Font Directories section of Chapter 27) must be set before fonts can be automatically loaded from files. DEFAULTPRINTINGHOST (see Chapter 29) should be set before attempting to generate hardcopy to a printer.

Network Systems — CH.DEFAULT.ORGANIZATION and CH.DEFAULT.DOMAIN (see the Name and Address Conventions section of Chapter 31) should be set to the default NS organization and domain, when using NS network communications. If CH.NET.HINT (see the Clearinghouse Functions section of Chapter 31) is set, it can reduce the amount of time spent searching for a clearinghouse.

Medley Executive — The variable PROMPT#FLG (see the Changing the Programmer's Assistant section of Chapter 13) determines whether an "event number" is printed at the beginning of every input line. The function CHANGESLICE (see the Changing the Programmer's Assistant section of Chapter 13) can be used to change the number of events that are remembered on the history list.

Copyright Notices — COPYRIGHTFLG, COPYRIGHTOWNERS, and DEFAULTCOPYRIGHTOWNER (see the Copyright Notices section of Chapter 17) control the inclusion of copyright notices on source files.

Printing Functions — **COMMENT**FLG (see the Comment Feature section of Chapter 26) determines how program comments are printed. FIRSTCOL, PRETTYFLG, and CLISPIFYPRETTYFLG (see the Special Prettyprint Controls section of Chapter 26) are among the many variables controlling how functions are pretty printed.

List Structure Editor — The variable INITIALSLST (see the Time Stamps section of Chapter 16) is used when "time-stamps" are inserted in a function when it is edited. EDITCHARACTERS (see the Time Stamps section of Chapter 16) is used to set the read macros used in the teletype editor.

## Idle Mode

The Medley environment runs on small single-user computers, usually located in users' offices. Often, users leave their computers up and running for days, which can cause several problems. First, the phosphor in the video display screen can be permanently marked if the same pattern is displayed for a long time (weeks). Second, if you go away, leaving a Medley system running, another person could possibly walk up and use the environment, taking advantage of any passwords that have been entered. To solve these problems, Medley implements the concept of "idle mode."

If no keyboard or mouse action has occurred for a specified time, Medley automatically enters idle mode. While idle mode is on, the display screen is blacked out, to protect the phosphor. Idle mode also runs a program to display some moving pattern on the black screen, so the screen does not appear to be broken. Usually, idle mode can be exited by pressing any key on the keyboard or mouse. However, you can optionally specify that idle mode should erase the current password cache when it is entered., and require the next user to supply a password to exit idle mode.

If either shift key is pressed while Medley is in idle mode, the current user name and the amount of time spent idling are displayed in the prompt window while the key is depressed.

Idle mode can also be entered by calling the function IDLE , or by selecting the Idle menu command from the background menu (see Chapter 28). The Idle menu command has subitems that allow you to interactively set the idle options (display program, erasing password, etc.) specified by the variable IDLE.PROFILE.

**IDLE.PROFILE** [Variable]

The value of this variable is a property list (see Chapter 3) which controls most aspects of idle mode. The following properties are recognized:

TIMEOUT Value is a number that determines how long (in minutes) Medley will wait before automatically entering idle mode. If NIL, idle mode will never be entered automatically. Default is 10 minutes.

FORGET If this is the symbol FIRST, your password will be erased when idle mode is entered. If non-NIL, your password will be erased when idle mode is exited. Initial value is T (erase password on exit).

If the password is erased on entry to idle mode (value FIRST), any programs left running when idle mode is entered will fail if they try doing anything requiring passwords (such as accessing file servers).

ALLOWED.LOGINS The value of this property can either be a list or a non-list. If the value is NIL or any other non-list, idle mode is exited without requesting login.

If the value is a list the members of the list should be interpreted as follows:

* If the value is a list containing * as it's element, login is required but anyone can exit idle mode. This will overwrite the previous user's user name and password each time idle mode is exited.

T Let the previous user (as determined by USERNAME) exit idle mode. If the username has not been set, this is equivalent to *

user name Let this specific user exit idle mode.

group name Let any member of this group (an NS clearinghouse group) exit idle mode.

AUTHENTICATE The value of this property determines the method used for logging in. The value can be one of the following:

T or NS Use the NS authentication protocol. This requires that you have an NS authentication server accessible on your net.

GV Authenticate the login via the GrapeVine protocol.

UNIX Use the unix login mechanism.

**Note**: Unix is case sensitive. If you try to login but fail, you may have typed the password with the caps-lock on.

LOGIN.TIMEOUT This is the number of seconds idle will wait for a login before resuming idle mode again.

DISPLAYFN The value of this property, which should be a function name or lambda expression, is called to display a moving pattern on the screen while in idle mode. This function is called with one argument, a window covering the whole screen. The default is IDLE.BOUNCING.BOX (below).

Any function used as a DISPLAYFN should call BLOCK (see Chapter 23) frequently, so other programs can run during idle mode.

SAVEVM Value is a number that determines how long (in minutes) after idle mode is entered that SAVEVM will be called to save the virtual memory. If NIL, SAVEVM is never called automatically from idle mode. Default is 10 minutes.

SUSPEND.PROCESS.NAMES Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

**IDLE.RESETVARS** [Variable]

The value of this variable is a list of two-element lists: ((*VAR*$_1$ *EXP*$_1$)(*VAR*$_2$ *EXP*$_2$)...). On entering idle mode, each variable *VAR*$_N$ is bound to the value of the corresponding expression *EXP*$_N$. When idle mode is exited, each variable *VAR*$_N$ is reset to its original value.

**IDLE.SUSPEND.PROCESS.NAMES** [Variable]

Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

**IDLE.PROFILE** [Variable]

The value of this variable determines the menu raised by selecting the Display subitem of the Idle background menu command. It should be in the format used for the ITEMS field of a menu (see Chapter 28), with the selection of an item returning the appropriate display function.

(**IDLE.BOUNCING.BOX** *WINDOW BOX WAIT*) [Variable]

This is the default display function used for idle mode. *BOX* is bounded about *WINDOW*, with bounces taking place every *WAIT* milliseconds. *BOX* can be a string, a bitmap, a window (whose image will be bounced about), or a list containing any number of these

(which will be cycled through). *BOX* defaults to the value of the variable IDLE.BOUNCING.BOX, which is initially a bitmap of the Venue logo. *WAIT* defaults to 1000 (one second).

## Saving Virtual Memory State

Medley storage allocation occurs within a virtual memory space that is usually much larger than the physical memory on the computer. The virtual memory is stored as a large file on the computer's hard disk, called the virtual memory file. Medley controls the swapping of pages between this file and the real memory, swapping in virtual memory pages as they are accessed, and swapping out pages that have been modified. At any moment, the total state of the Medley virtual memory is stored partially in the virtual memory file, and partially in the real physical memory.

Medley provides facilities for saving the total state of the virtual memory, either on the virtual memory file, or in a file on an arbitrary file device. The function LOGOUT is used to write all altered (dirty) pages from the real memory to the virtual memory file and stop Medley, so that Medley can be restarted from the state of the LOGOUT. SAVEVM updates the virtual memory file without stopping Medley, which puts the virtual memory file into a consistant state (temporarily), so it could be restarted if the system crashes. SYSOUT and MAKESYS are used to save a copy of the total virtual memory state on a file, which can be loaded into another machine to restore Medley's state. VMEM.PURE.STATE can be used to "freeze" the current state of the virtual memory, so that Medley will come up in that state if it is restarted.

(**LOGOUT** *FAST*) [Function]

Stops Medley, and returns control to the operating system. If Medley is restarted, it should come up in the same state as when the LOGOUT was called. LOGOUT will not affect the state of open files.

LOGOUT writes out all altered pages from real memory to the virtual memory file. If *FAST* is T, Medley is stopped without updating the virtual memory file. Note that after doing (LOGOUT T) it will not be possible to restart Medley from the point of the LOGOUT, and it may not be possible to restart it at all. Typing (LOGOUT T**)** is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

If *FAST* is the symbol ?, LOGOUT acts like *FLG* = T if the virtual memory file is consistant, otherwise it acts like *FLG* = NIL. This insures that the virtual memory image can be restarted as of *some* previous state, not necessarily as of the LOGOUT.

(**SAVEVM** —) [Function]

This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should there be a system crash or other disaster.

If the system has been idle long enough (no keyboard or mouse activity), there are dirty pages to be written, and there are few enough dirty pages left to write that a SAVEVM would be quick, SAVEVM is automatically called. When SAVEVM is called automatically,

the cursor is changed to a special cursor:  , stored in the variable SAVINGCURSOR. You can control how often SAVEVM is automatically called by setting the following two global variables:

**SAVEVMWAIT** [Variable]
**SAVEVMMAX** [Variable]

The system will call SAVEVM after being idle for SAVEVMWAIT seconds (initially 300) if there are fewer than SAVEVMMAX pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set SAVEVMWAIT = 0 and SAVEVMMAX = 10000, in which case SAVEVM will be called the first chance available after the first dirty page has been written.

The function SYSOUT saves the current state of Medley's virtual memory on a file, known as a "sysout file", or simply a "sysout". The file package can be used to save particular function definitions and other arbitrary objects on files, but SYSOUT saves the *total* state of the system. This capability can be useful in many situations: for creating customized systems for other people to use, or to save a particular system state for debugging purposes. Note that a sysout file can be very large (thousands of pages), and can take a long time to create, so it is not to be done lightly. The file produced by SYSOUT can be loaded into Medley's virtual memory and restarted to restore the virtual memory to the exact state that it had when the sysout file was made. The exact method of loading a sysout depend on the implementation. For more information on loading sysout files, see the users guide for your computer.

(**SYSOUT** *FILE*) [Function]

Saves the current state of Medley's virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so (PROGN (SYSOUT 'FOO) (PRINT 'HELLO)) will cause HELLO to be printed after the sysout file is restarted.

SYSOUT can take a very long time (ten or fifteen minutes), particularly when storing a file on a remote file server. To display some indication that something is happening, the cursor is changed to:  Also, as the sysout file is being written, the cursor is inverted line by line, to show that activity is taking place, and how much of the sysout has completed. For example, after the SYSOUT is about two-thirds done, the cursor would look like:  . The SYSOUT cursor is stored in the variable SYSOUTCURSOR.

If *FILE* is non-NIL, the variable SYSOUTFILE is set to the body of *FILE*. If *FILE* is NIL, then the value of SYSOUTFILE instead. Therefore, (SYSOUT) will save the current state on the next higher version of a file with the same name as the previous SYSOUT. Also, if the extension for *FILE* is not specified, the value of SYSOUT.EXT is used. SYSOUT sets SYSOUTDATE (see the System Version Information section below) to (DATE), the time and date that the SYSOUT was performed.

If SYSOUT was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, SYSOUT returns NIL. Otherwise it returns the full file name of *FILE*.

Actually, SYSOUT "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, SYSOUT returns a list whose CAR is the full file name of *FILE*. For example, (if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO)) will cause HELLO to be printed when the sysout file is restarted, but not when SYSOUT is initially performed.

**Note**: SYSOUT does not save the state of any open files. Use WHENCLOSE (see the Closing and Reopening Files section in Chapter 24) to associate certain operations with open files so that when a SYSOUT is started up, these files will be reopened, and file pointers repositioned.

SYSOUT evaluates the expressions on BEFORESYSOUTFORMS (see also AROUNDEXITFNS) before creating the sysout file. This variable initially includes expressions to:

1. Set the variables SYSOUTDATE and SYSOUTFILE as described above
2. Default the sysout file name *FILE* according to the values of the variables SYSOUTFILE and SYSOUT.EXT, as described above
3. Perform any necessary operations on open files as specified by calls to WHENCLOSE.

After a sysout file is restarted (but *not* when it is initially created), SYSOUT evaluates the expressions on AFTERSYSOUTFORMS (see also AROUNDEXITFNS). This initially includes expressions to:

1. Perform any necessary operations on previously-opened files as specified by calls to WHENCLOSE
2. Possibly print a message, as determined by the value of SYSOUTGAG (see below)
3. Call SETINITIALS to reset the initials used for time-stamping (see the Time Stamps section of Chapter 16).

**AROUNDEXITFNS** [Variable]

This variable provides a way to "advise" the system on cleanup and restoration activities to perform around LOGOUT, SYSOUT, MAKESYS and SAVEVM; It subsumes the functionality of BEFORESYSOUTFORMS, AFTERLOGOUTFORMS, etc. It's value is a list of functions (names) to call around every "exit" of the system. Each function is called with one argument, a symbol indicating which particular event is occuring. The symbols are:

**BEFORLOGOUT** The system is about to perform a LOGOUT.

**BEFORESYSOUT**
**BEFOREMAKESYS**
**BEFORESAVEVM** The system is about to perform a SYSOUT, MAKESYS or a SAVEVM.

**AFTERLOGOUT**
**AFTERSYSOUT**
**AFTERMAKESYS**
**AFTERSAVEVM** The system is starting up am image that was saved by performing a LOGOUT, SYSOUT, etc.

**AFTERDOSYSOUT**

**AFTERDOMAKESYS**
**AFTERDOSAVEVM** The system just made a copy of the virtual memory and saved it to disk. The image continues to run. These events only exist to allow you to negate the effects of saveing a copy of the virtual memory.

**SYSOUTGAG** [Variable]

The value of SYSOUTGAG determines what is printed when a sysout file is restarted. If the value of SYSOUTGAG is a list, the list is evaluated, and no additional message is printed. This allows you to print a message. If SYSOUTGAG is non-NIL and not a list, no message is printed. Finally, if SYSOUTGAG is NIL (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, you are greeted by printing the value of HERALDSTRING (see below) followed by a greeting message. If the sysout file was made by a different user, a message is printed, warning that the currently-loaded user init file may be incorrect for the current user (see the Greeting and Initialization Files section above).

(**MAKESYS** *FILE NAME*) [Function]

Used to store a new Medley system on the "makesys file" *FILE*. Like SYSOUT, but before the file is made, the system is "initialized" by undoing the greet history, and clearing the display.

When the system is first started up, a "herald" is printed identifying the system, typically "Medley-XX DATE ...". If *NAME* is non-NIL, MAKESYS will use it instead of Medley-XX in the herald. MAKESYS sets HERALDSTRING to the herald string printed out.

MAKESYS also sets the variable MAKESYSDATE (see the next section below) to (DATE), i.e. the time and date the system was made.

Medley contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into the virtual memory file (making it non-continuable). The frequency with which this routine runs is determined by:

**BACKGROUNDPAGEFREQ** [Variable]

This variable determines how often the routine that writes out dirty pages is run. The *higher* BACKGROUNDPAGEFREQ is set, the *greater* the time between running the dirty page writing routine. Initially it is set to 4. The lower BACKGROUNDPAGEFREQ is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.

(**VMEM.PURE.STATE** *X*) [NoSpread Function]

VMEM.PURE.STATE modifies the swapper's page replacement algorithm so that dirty pages are only written at the end of the virtual memory backing file. This "freezes" a given virtual memory state, so that Medley will come up in that state whenever it is restarted. This can be used to set up a "clean" environment on a pool machine, allowing each user to initialize the system simply by rebooting the computer.

The way to use VMEM.PURE.STATE is to set up the environment as you wish it to be "frozen," evaluate (VMEM.PURE.STATE T), and then call any function that saves the virtual memory state (LOGOUT, SAVEVM, SYSOUT, or MAKESYS). From that point on,

whenever the system is restarted, it will return to the state as of the saving operation. Future LOGOUT, SAVEVM, etc. operations will not reset this state.

**Note**: When the system is running in "pure state" mode, it uses a significant amount of the virtual memory backing file to save the "frozen" memory image, so this will reduce the amount of virtual memory space available for use.

(VMEM.PURE.STATE) returns T if the system is running in "pure state" mode, NIL otherwise.

(**REALMEMORYSIZE**) [Function]

Returns the number of real memory pages in the computer.

(**VMEMSIZE**) [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk (see SYSOUT, above).

**LASTVMEMFILEPAGE** [Variable]

Value is the total size of the virtual memory backing file. This variable is set when the system is started. You should *not* set it..

**Note**: When the virtual memory expands to the point where the virtual memory backing file is almost full, a break will occur with the warning message "Your virtual memory backing file is almost full. Save your work & reload asap." When this happens, it is strongly suggested that you save any important work and reload the system. If you continue working past this point, the system will start slowing down considerably, and it will eventually stop working.

## System Version Information

Medley runs on a number of different machines, with many possible hardware configurations. There have been a number of different releases of the Medley software. These facts make it difficult to answer the important question "what software/hardware environment are you running?" when reporting bugs. The following functions allow the novice to collect this information.

(**PRINT-LISP-INFORMATION** *STREAM FILESTRING*) [NoSpread Function]

Prints out a summary of the software and hardware environment that Medley is running in, and a list of all loaded patch files:

```
Venue Medley version
Medley 2.0 sysout of 7-Oct-92 15:18:52 on mips,
Emulator created: 20-Nov-92, memory size: 0,
machine d022899 mo
based on Envos Medley version Medley 2.0 sysout of 7-Oct-
92 15:18:52,
Make-init dates: 7-Oct-92 11:07:17, 7-Oct-92 11:26:22
Patch files: NIL
```

*STREAM* is the stream used to print the summary. If not given, it defaults to T.

*FILESTRING* is a string used to determine what loaded files should be listed as "patch files." All file names on LOADEDFILELST (see the Noticing Files section of Chapter 17) that have *FILESTRING* as a substring as listed. If *FILESTRING* is not given, it defaults to the string "PATCH".

(**CL:LISP-IMPLEMENTATION-TYPE**) [Function]

Returns a string identifying the type of implementation that is running, e.g., "Medley".

(**CL:LISP-IMPLEMENTATION-VERSION**) [Function]

Returns a string identifying the version that is running. Currently gives the system name and date, e.g., "KOTO of 10-Sep-85 08:25:46".

This uses the variables MAKESYSNAME and MAKESYSDATE (below), so it will change if you use MAKESYS (see the Saving Virtual Memory State section above) to create a custom sysout file, or explicitly changes these variables.

(**CL:SOFTWARE-TYPE**) [Function]

Returns a string identifying the operating system that Interlisp is running under. Currently returns the string "Envos Medley".

(**CL:SOFTWARE-VERSION**) [Function]

Returns a string identifying the version of the operating system that Interlisp is running under. Currently, this returns the date that the Medley release was originally created, so it doesn't change over MAKESYS or SYSOUT.

(**CL:MACHINE-TYPE**) [Function]

Returns a string identifying the type of computer hardware that Medley is running on, i.e., "1108", "1132", "1186", "mips", etc.

(**CL:MACHINE-VERSION**) [Function]

Returns a string identifying the version of the computer hardware that Medley is running on. Currently returns the microcode version and real memory size.

(**CL:MACHINE-INSTANCE**) [Function]

Returns a string identifying the particular machine that Medley is running on. Currently returns the machine's NS address.

(**CL:SHORT-SITE-NAME**) [Function]

Returns a short string identifying the site where the machine is located. Currently returns (ETHERHOSTNAME) (if non-NIL) or the string "unknown".

(**CL:LONG-SITE-NAME**) [Function]

Returns a long string identifying the site where the machine is located. Currently returns the same as SHORT-SITE-NAME.

**SYSOUTDATE** [Variable]

Value is set by SYSOUT (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

**MAKESYSDATE** [Variable]

Value is set by MAKESYS (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

**MAKESYSNAME** [Variable]

Value is a symbol identifying the release name of the current Medley system, e.g., :MEDLEY.

(**SYSTEMTYPE**) [Function]

Allows programmers to write system-dependent code. SYSTEMTYPE returns a symbol corresponding to the implementation of Interlisp: D (for Medley), TOPS-20, TENEX, JERICO, or VAX.

In Medley, (SELECTQ (SYSTEMTYPE) ...) expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(**MACHINETYPE**) [Function]

Returns the type of machine that Medley is running on: either DORADO (for the Xerox 1132), DOLPHIN (for the Xerox 1100), DANDELION (for the Xerox 1108), DOVE (for the Xerox 1186), or MAIKO (for Unix, DOS, etc).

## Date And Time Functions

(**DATE** *FORMAT*) [Function]

Returns the current date and time as a string with format "*DD-MM-YY HH:MMM:SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., "7-Jun-85 15:49:34".

If *FORMAT* is a date format as returned by DATEFORMAT (below), it is used to modify the format of the date string returned by DATE.

(**IDATE** *STR*) [Function]

*STR* is a date and time string. IDATE returns *STR* converted to a number such that if DATE1 is before (earlier than) DATE2, then (IDATE *DATE1*) < (IDATE *DATE2*). If STR is NIL, the current date and time is used.

Different Interlisp implementations can have different internal date formats. However, IDATE always has the essential property that (IDATE *X*) is less than (IDATE *Y*) if *X* is before *Y*, and (IDATE (GDATE *N*)) equals *N*. Programs which do arithmetic other than numerical comparisons between IDATE numbers may not work when moved from one implementation to another.

Generally, it is possible to increment an IDATE number by an integral number of days by computing a "1 day" constant, the difference between two convenient IDATEs, e.g. (IDIFFERENCE (IDATE "2-JAN-80 12:00") (IDATE "1-JAN-80 12:00")). This "1 day" constant can be evaluated at compile time.

IDATE is guaranteed to accept as input the dates that DATE will output. It will ignore the parenthesized day of the week (if any). IDATE also correctly handles time zone specifications for those time zones registered in the list TIME.ZONES (below).

(**GDATE** *DATE FRMAT* —) [Function]

Like DATE, except that *DATE* can be a number in internal date and time format as returned by IDATE. If *DATE* is NIL, the current time and date is used.

(**DATEFORMAT** $KEY_1$ . . . $KEY_N$) [NLambda NoSpread Function]

DATEFORMAT returns a date format suitable as a parameter to DATE and GDATE. $KEY_1$ . . . $KEY_N$ are a set of keywords (unevaluated). Each keyword affects the format of the date independently (except for SLASHES and SPACES). If the date returned by (DATE) with the default formatting was 7-Jun-85 15:49:34, the keywords would affect the formatting as follows:

| | |
|---|---|
| NO.DATE | Doesn't include the date information, e.g. "15:49:34". |
| NUMBER.OF.MONTH | Shows the month as a number instead of a name, e.g. "7-06-85 15:49:34". |
| YEAR.LONG | Prints the year using four digits, e.g. "7-Jun-1985 15:49:34". |
| SLASHES | Separates the day, month, and year fields with slashes, e.g. "7/Jun/85 15:49:34". |
| SPACES | Separates the day, month, and year fields with spaces, e.g. "7 Jun 85 15:49:34". |
| NO.LEADING.SPACES | By default, the day field will always be two characters long. If NO.LEADING.SPACES is specified, the day field will be one character for dates earlier than the 10th, e.g. "7-Jun-85 15:49:34" instead of "7-Jun-85 15:49:34". |
| NO.TIME | Doesn't include the time information, e.g. "7-Jun-85". |
| TIME.ZONE | Includes the time zone in the time specification, e.g. "7-Jun-85". |
| NO.SECONDS | Doesn't include the seconds, e.g. "7-Jun-85 15:49". |
| DAY.OF.WEEK | Includes the day of the week in the time specification, e.g. "7-Jun-85 15:49:34 PDT (Friday)". |

DAY.SHORT If DAY.OF.WEEK is specified to include the day of the week, the week day is shortened to the first three letters, e.g. "7-Jun-85 15:49:34 PDT (Fri)". Note that DAY.SHORT has no effect unless DAY.OF.WEEK is also specified.

(**CLOCK** *N* —) [Function]

If *N* = 0, CLOCK returns the current value of the time of day clock i.e., the number of milliseconds since last system start up.

If *N* = 1, returns the value of the time of day clock when you started up this Interlisp, i.e., difference between (CLOCK 0) and (CLOCK 1) is number of milliseconds (real time) since this Interlisp system was started.

If *N* = 2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

If *N* = 3, returns the number of milliseconds of compute time spent in garbage collections (all types).

(**SETTIME** *DT*) [Function]

Sets the internal time-of-day clock. If *DT* = NIL, SETTIME attempts to get the time from the communications net; if it fails, you are prompted for the time. If *DT* is a string in a form that IDATE recognizes, it is used to set the time.

The following variables are used to interpret times in different time zones. \TimeZoneComp, \BeginDST, and \EndDST are normally set automatically if your machine is connected to a network with a time server. For standalone machines, it may be necessary to set them by hand (or in your init file, see the first section of this chapter) if you are not in the Pacific time zone.

**TIME.ZONES** [Variable]

Value is an association list that associates time zone specifications (PDT, EST, GMT, etc.) with the number of hours west of Greenwich (negative if east). If the time zone specification is a single letter, it is appended to "DT" or "ST" depending on whether daylight saving time is in effect. Initially set to:

```
((8 . P) (7 . M) (6 . C) (5 . E) (0 . GMT))
```

This list is used by DATE and GDATE when generating a date with the TIME.ZONE format is specified, and by IDATE when parsing dates.

**\TimeZoneComp** [Variable]

This variable should be initialized to the number of hours west of Greenwich (negative if east). For the U.S. west coast it is 8. For the east coast it is 5.

**\BeginDST** [Variable]
**\EndDST** [Variable]

\BeginDST is the day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day); \EndDST is the day on or before which Daylight Savings Time ends. Days are numbered with 1 being January 1, and

counting the days as for a leap year. In the USA where Daylight Savings Time is observed, \BeginDST = 121 and \EndDST = 305. In a region where Daylight Savings Time is not observed at all, set \BeginDST to 367.

## Timers and Duration Functions

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarm clock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Medley environment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

Three functions are provided: SETUPTIMER, SETUPTIMER.DATE, and TIMEREXPIRED?. There are also several new i.s.oprs: forDuration, during, untilDate, timerUnits, usingTimer, and resourceName (reasonable variations on upper/lower case are permissible).

These functions use an object called a timer, which encodes a future clock time at which a signal is desired. A timer is constructed by the functions SETUPTIMER and SETUPTIMER.DATE, and is created with a basic clock "unit" selected from among SECONDS, MILLISECONDS, or TICKS. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is MILLISECONDS.

Currently, the TICKS unit depends on what machine Medley is running on. The Xerox 1132 has about 1680 ticks per millisecond; the Xerox 1108 has about 34.746 ticks per millisecond; the Xerox 1185 and 1186 have about 62.5 ticks per millisecond. The advantage of using TICKS rather than one of the uniform interfaces is primarily speed; e.g., it may take over 400 microseconds to read the milliseconds clock (a software facility that uses the real clock), whereas reading the real clock itself may take less than ten microseconds. The disadvantage of the TICKS unit is its short roll-over interval (about 20 minutes) compared to the MILLISECONDS roll-over interval (about two weeks), and also the dependency on particular machine parameters.

(**SETUPTIMER** *INTERVAL OldTimer? timerUnits intervalUnits*) [Function]

SETUPTIMER returns a timer that will "go off" (as tested by TIMEREXPIRED?) after a specified time-interval measured from the current clock time. SETUPTIMER has one required and three optional arguments:

*INTERVAL* must be a integer specifying how long an interval is desired. *timerUnits* specifies the units of measure for the interval (defaults to MILLISECONDS).

If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer. *intervalUnits* specifies the units in which the *OldTimer?* is expressed (defaults to the value of *timerUnits*.

(**SETUPTIMER.DATE** *DTS OldTimer?*) [Function]

SETUPTIMER.DATE returns a timer (using the SECONDS time unit) that will "go off" at a specified date and time. *DTS* is a Date/Time string such as IDATE accepts (see the above section). If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer.

SETUPTIMER.DATE operates by first subtracting (IDATE) from (IDATE *DTS*), so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(**TIMEREXPIRED?** *TIMER ClockValue.or.timerUnits*) [Function]

If *TIMER* is a timer, and *ClockValue.or.timerUnits* is the time-unit of *TIMER*, TIMEREXPIRED? returns true if *TIMER* has "gone off".

*ClockValue.or.timerUnits* can also be a timer, in which case TIMEREXPIRED? compares the two timers (which must be in the same timer units). If X and Y are timers, then (TIMEREXPIRED? X Y) is true if X is set for an *earlier* time than Y.

There are a number of i.s.oprs that make it easier to use timers in iterative statements (see the Interative Statement section of Chapter 9). These i.s.oprs are given below in the "canonical" form, with the second "word" capitalized, but the all-caps and all-lower-case versions are also acceptable.

**forDuration** *INTERVAL* [I.S. Operator]
**during** *INTERVAL* [I.S. Operator]

*INTERVAL* is an integer specifying an interval of time during which the iterative statement will loop.

**timerUnits** *UNITS* [I.S. Operator]

*UNITS* specifies the time units of the *INTERVAL* specified in forDuration.

**untilDate** *DTS* [I.S. Operator]

*DTS* is a Date/Time string (such as IDATE accepts) specifying when the iterative statement should stop looping.

**usingTimer** *TIMER* [I.S. Operator]

If usingTimer is given, *TIMER* is reused as the timer for forDuration or untilDate, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

**resourceName** *RESOURCE* [I.S. Operator]

*RESOURCE* specifies a resource name to be used as the timer storage (see the File Package Types section of Chapter 17). If *RESOURCE* = T, it will be converted to an internal name.

Some examples:

```
(during 6MONTHS timerUnits 'SECONDS
  until (TENANT-VACATED? HouseHolder)
  do (DISMISS <for-about-a-day>)
     (HARRASS HouseHolder)
  finally (if (NOT (TENANT-VACATED? HouseHolder))
              then (EVICT-TENANT HouseHolder)))
```

This example shows that how is is possible to have two termination condition: when the time interval of 6MONTHS has elapsed, or when the predicate (TENANT-VACATED? HouseHolder) becomes true. Note that the "finally" clause is executed regardless of which termination condition caused it.

Also note that since the millisecond clock will "roll over" about every two weeks, "6MONTHS" wouldn't be an appropriate interval if the timer units were the default case, namely MILLISECONDS.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
      do (CARRY.ON.AS.USUAL)
      finally (PROMPTPRINT "Have you had your 10-day check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
 (untilDate "31-DEC-83 23:59:59"  usingTimer \RandomTimer
  when (WINNING?) do (RETURN)
  finally (ERROR "You've been losing this whole year!"))
```

Here is a usage of an explicit date for the time interval; also, some stsorage has been squirreled away (as the value of \RandomTimer) for use by the call to SETUPTIMER in this loop.

```
(forDuration SOMEINTERVAL
  resourceName \INNERLOOPBOX
  timerunits 'TICKS
  do (CRITICAL.INNER.LOOP))
```

For this loop, you don't want any CONSing to take place, so \INNERLOOPBOX is defined as a resource which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a WITH-RESOURCES call. Furthermore, a time unit of TICKS is specified, for lower overhead in this critical inner loop. In fact specifying a resourceName of T is the same as specifying it to be \ForDurationOfBox; this is just a simpler way to specify that a resource is wanted, without having to think up a name.

## Resources

Medley is based on the use of a storage-management system which allocates memory space for new data objects, and automatically reclaims the space when no longer in use. More generally, Medley manages shared "resources", such as files, semaphors for processes, etc. The protocols for allocating and freeing such resources resemble those of ordinary storage management.

Sometimes you need to explicitly manage the allocation of resources. You may want the efficiency of explicit reclamation of certain temporary data; or it may be expensive to initialize a complex data object; or there may be an application that must not allocate new cells during some critical section of code.

The file manager type RESOURCES is available to help with the definition and usage of such classes of data; the definition of a RESOURCE specifies prototype code to do the basic management operations. The file manager command RESOURCES is used to save such definitions on files, and INITRESOURCES (see the Miscellaneous File Manager Commands section of Chapter 17) causes the initialization code to be output.

The basic needs of resource management are:

1. Obtaining a data item from the Lisp memory management system and configuring it to be a totally new instance of the resource in question
2. Freeing up an instance which is no longer needed

3. Getting an instance of the resource for temporary usage (whether "fresh" or a formerly freed-up instance)
4. Setting up any prerequisite global data structures and variables

A resources definition consists of four "methods": INIT, NEW, GET, and FREE; each "method" is a form that will specialize the definition for four corresponding user-level macros INITRESOURCE, NEWRESOURCE, GETRESOURCE, and FREERESOURCE. PUTDEF is used to make a resources definition, and the four components are specified in a proplist:

```
(PUTDEF
    'RESOURCENAME
    'RESOURCES
    '(NEW  NEW-INSTANCE-GENERATION-CODE
      FREE FREEING-UP-CODE
      GET  GET-INSTANCE-CODE
      INIT INITIALIZATION-CODE))
```

Each of the *xxx-CODE* forms is a form that will appear as if it were the body of a substitution macro definition for the corresponding macro (see the discussion on the macros below).

### A Simple Example

Suppose one has several pieces of code which use a 256-character string as a scratch string. One could simply generate a new string each time, but that would be inefficient if done repeatedly. If you can guarantee that there are no re-entrant uses of the scratch string, then it could simply be stored in a global variable. However, if the code might be re-entrant on occasion, the program has to take precautions that two programs do not use the same scratch string at the same time. (This consideration becomes very important in a multi-process environment. It is hard to guarantee that two processes won't be running the same code at the same time, without using elaborate locks.) A typical tactic would be to store the scratch string in a global variable, and set the variable to NIL whenever the string is in use (so that re-entrant usages would know to get a "new" instance). For example, assuming the global variable TEMPSTRINGBUFFER is initialized to NIL:

```
[DEFINEQ (WITHSTRING NIL
   (PROG ((BUF (OR (PROG1 TEMPSTRINGBUFFER
                (SETQ TEMPSTRINGBUFFER NIL))
           (ALLOCSTRING 256))))
```

... use the scratch string in the variable BUF ...

```
          (SETQ TEMPSTRINGBUFFER BUF)
           (RETURN]
```

Here, the basic elements of a "resource" usage may be seen:

1. A call (ALLOCSTRING 256) allocates fresh instances of "buffer"
2. After usage is completed the instance is returned to the "free" state, by putting it back in the global variable TEMPSTRINGBUFFER where a subsequent call will find it
3. The prog-binding of BUF will get an existing instance of a string buffer if there is one -- otherwise it will get a new instance which will later be available for reuse
4. Some initialization is performed before usage of the resource (in this case, it is the setting of the global variable TEMPSTRINGBUFFER).

Given the following resources definition:

```
(PUTDEF
    'STRINGBUFFER
    'RESOURCES
    '(NEW  (ALLOCSTRING 256)
      FREE (SETQ TEMPSTRINGBUFFER (PROG1 . ARGS))
      GET  (OR (PROG1 TEMPSTRINGBUFFER
                      (SETQ TEMPSTRINGBUFFER NIL))
               (NEWRESOURCE TEMPSTRINGBUFFER)))
      INIT (SETQ TEMPSTRINGBUFFER NIL)))
```

we could then redo the example above as

```
(DEFINEQ (WITHSTRING NIL
    (PROG ((BUF (GETRESOURCE STRINGBUFFER)))
```

... use the string in the variable BUF ...

```
      (FREERESOURCE STRINGBUFFER BUF)
       (RETURN]
```

The advantage of doing the coding this way is that the resource management part of WITHSTRING is fully contained in the expansions of GETRESOURCE and FREERESOURCE, and thus there is no confusion between what is WITHSTRING code and what is resource management code. This particuar advantage will be multiplied if there are other functions which need a "temporary" string buffer; and of course, the resultant modularity makes it much easier to contemplate minor variations on, as well as multiple clients of, the STRINGBUFFER resource.

In fact, the scenario just shown above in the WITHSTRING example is so commonly useful that an abbreviation has been added; if a resources definition is made with *only* a NEW method, then appropriate FREE, GET, and INIT methods will be inferred, along with a coordinated globalvar, to be parallel to the above definition. So the above definition could be more simply written

```
(PUTDEF 'STRINGBUFFER
         'RESOURCES
         '(NEW  (ALLOCSTRING 256)))
```

and everything would work the same.

The macro WITH-RESOURCES simplifies the common scoping case, where at the beginning of some piece of code, there are one or more GETRESOURCE calls the results of which are each bound to a lambda variable; and at the ending of that code a FREERESOURCE call is done on each instance. Since the resources are locally bound to variables with the same name as the resource itself, the definition for WITHSTRING then simplifies to

```
(DEFINEQ (WITHSTRING NIL
    (WITH-RESOURCES (STRINGBUFFER)
```

... use the string in the variable STRINGBUFFER ...]

### Trade-offs in More Complicated Cases

This simple example presumes that the various functions which use the resource are generally not re-entrant. While an occasional re-entrant use will be handled correctly (another example of the resource will simply be created), if this were to happen too often, then many of the resource requests will create and throw away new objects, which defeats one of the major purposes of using resources. A slightly more complex GET and FREE method can be of much benefit in maintaining a pool of available

resources; if the resource were defined to maintain a list of "free" instances, then the GET method could simply take one off the list and the FREE method could just push it back onto the list. In this simple example, the SETQ in the FREE method defined above would just become a "push", and the first clause of the GET method would just be (pop TEMPSTRINGBUFFER)

A word of caution: if the datatype of the resource is something very small that Medley is "good" at allocating and reclaiming, then explicit user storage management will probably not do much better than the combination of cons/createcell and the garbage collector. This would especially be so if more complicated GET and FREE methods were to be used, since their overhead would be closer to that of the built-in system facilities. Finally, it must be considered whether retaining multiple instances of the resource is a net gain; if the re-entrant case is truly rare, it may be more worthwhile to retain at most one instance, and simply let the instances created by the rarely-used case be reclaimed in the normal course of garbage collection.

### Macros for Accessing Resources

Four user-level macros are defined for accessing resources:

(**NEWRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**FREERESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**GETRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]
(**INITRESOURCE** *RESOURCENAME* . *ARGS*) [Macro]

Each of these macros behave as if they were defined as a substitution macro of the form

((*RESOURCENAME* . *ARGS*) *MACROBODY*)

where the expression *MACROBODY* is selected by using the "code" supplied by the corresponding method from the *RESOURCENAME* definition.

Note that it is possible to pass "arguments" to your resource allocation macros. For example, if the GET method for the resource FOO is (GETFOO . ARGS), then (GETRESOURCE FOO X Y) is transformed into (GETFOO X Y). This form was used in the FREE method of the STRINGBUFFER resource described above, to pass the old STRINGBUFFER object to be freed.

(**WITH-RESOURCES** (*RESOURCE*$_1$ *RESOURCE*$_2$ ...) *FORM*$_1$ *FORM*$_2$ ...) [Macro]

The WITH-RESOURCES macro binds lambda variables of the same name as the resources (for each of the resources *RESOURCE*$_1$, *RESOURCE*$_2$, etc.) to the result of the GETRESOURCE macro; then executes the forms *FORM*$_1$, *FORM*$_2$, etc., does a FREERESOURCE on each instance, and returns the value of the last form (evaluated and saved before the FREERESOURCEs).

**Note**: (WITH-RESOURCES *RESOURCE* ...) is interpreted the same as (WITH-RESOURCES (*RESOURCE*) ...). Also, the singular name WITH-RESOURCE is accepted as a synonym for WITH-RESOURCES.

### Saving Resources in a File

Resources definitions may be saved on files using the RESOURCES file package command (see the Miscellaneous File Package Commands section of Chapter 17). Typically, one only needs the full definition available when compiling or interpreting the code, so it is appropriate to put the file package command in a (DECLARE: EVAL@COMPILE DONTCOPY ...) declaration, just as one might

do for a RECORDS declaration. But just as certain record declarations need *some* initialization in the run-time environment, so do most resources. This initialization is specified by the resource's INIT method, which is executed automatically when the resource is defined by the PUTDEF output by the RESOURCES command. However, if the RESOURCES command is in a DONTCOPY expression and thus is not included in the compiled file, then it is necessary to include a separate INITRESOURCES command (see the Miscellaneous File Manager Commands section of Chapter 17) in the filecoms to insure that the resource is properly initialized.

[This page intentionally left blank]

MISCELLANEOUS