Warp Programming Environment: User Manual

Bernd Bruegge

20 January 1988

CMU-CS-88-105 (R)

Abstract

The goal of the Warp Programming Environment is to provide easy access to the Warp machine, a parallel supercomputer based on the systolic array architecture. The Warp Programming Environment offers a uniform environment for editing, compiling, debugging and executing Warp programs. It is based on an extensible shell written in Common Lisp and a runtime system written in C. It runs on a SUN-3 workstation under UNIX 4.2 and supports three types of users: Implementors of the environment itself who modify or enhance the functionality of the environment. Developers using the compiler and debugger to write Warp programs. And programmers writing application programs that call Warp programs. This document describes the Warp Programming Environment 2.6 from 08-Jan-88.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

Table of Contents

List of Figures

1. Introduction

The basic philosophy of the Warp Programming Environment (WPE) is to provide easy access to the Warp machine, replacing the complexity of earlier programming environments for Warp. Warp is a low-cost supercomputer based on the systolic array architecture[1]. The Warp Programming Environment offers the user the ability to compile, execute and debug W2 programs[2] on the Warp machine in a uniform environment. Commands in the Warp Programming Environment are provided in a uniform way based on an extensible shell. This shell is called the *Warp shell.*

The Warp shell has several attractive features. First of all, it provides a uniform help mechanism. An example from the help description of a command can be fed to the command interpreter, providing an easy exploration of the command language. Second, it takes care of different levels of sophistication, in particular of the novice and the experienced user; this can often be the same person at different times during the program development. The underlying Common Lisp implementation and the components of the environment are completely hidden from the application programmer; this is useful for somebody who is just interested in using the Warp shell. However, the Lisp implementation and all the software components comprising the Warp environment are easily available if so desired. This makes the Warp shell extensible. In particular, an interested programmer can make use of Common Lisp's powerful control structures to implement new commands.

The Warp shell is designed to run inside a text editor. The advantage of using an editor is that features such as intra-line editing, history buffers, re-execution of commands of previous commands and creation of script files are automatically available without any additional cost.

The Warp Programming Environment supports *multiple user access*, because the use of the Warp machine in a typical user session is sporadic. And it supports *multiple machine access*: If there is more than one Warp machine available, the user has the choice of connecting to any of these machines.

The Warp Programming Environment provides *network transparency* which is achieved by *servers* communicating with Warp shells running on remote workstations via remote procedure calls using the TCP-IP protocol. This means that the Warp machine can be accessed from any site that understands the TCP-IP protocol.

Each server is based on the *Warp monitor*, a software package designed to provide a "virtual Warp machine". The main goal of the Warp monitor is to shield the programmer from the complexity of the Warp array and yet make the hardware accessible. The Warp monitor contains functions for locking the machine, allocation of memory, for the execution and for the inspection of data and code.

In addition to providing a server for the Warp shell, the Warp monitor can also be used by programmers who want to call W2 programs from their own application programs. This includes applications running inside the Warp shell, for example debugging tools for the Warp machine as well as standalone applications. A package called the *Warp User Package*[3] has been implemented on top of the Warp monitor that enables the application programmer to write standalone applications using Warp programs from a library without having to know any details about the Warp machine at all.

---

[1]Marco Annaratone, Emmanuel Arnould, Robert Cohn, Thomas Gross, H.T. Kung, Monica Lam, Onat Menzilcioglu, Ken Sarocky, John Senko, Jon Webb, "Architecture of Warp", Compcon Spring 1987, IEEE Computer Society, 1987.

[2]Thomas Gross, Monica Lam, *"A Description of W2"*, Warp Document D330R05, Department of Computer Science, Carnegie-Mellon University, September 10, 1986.

[3]Mosur Ravishankar, *"Warp User Package"*, Warp Document D150R00, Department of Computer Science, Carnegie-Mellon University, March 1987.

As a result of this structure, the Warp Programming Environment supports three types of users: The implementor of the environment itself, who can use the Warp shell to define the functionality of the environment. Developers of W2 programs, who can use the W2 compiler and W2 debugger to develop W2 programs. And application programmers who can write standalone programs calling W2 programs provided by a library.

1.1 How to get the Warp Programming Environment

The Warp Programming Environment consists of several software components. The file system location of these components is defined by a set of environment variables in the socalled *environment file* **warprc**. The location of the **warprc** file is system dependent and is usually stored in the WPE home directory. Ask your system maintainer for the value of the environment variable `$WPEhome`. Versions of WPE can be found in the directories `$WPEhome/current` or `$WPEhome/exp`, respectively. The former one contains a stable version and the latter one a more experimental version.

WPE is designed to be executable in several ways: as a standalone shell, inside a text editor and inside a window manager. Currently two Emacs editors are supported: GNU Emacs[4] and Gosling's Emacs[5].

To install WPE under GNU Emacs, add the following lines to your `.emacs` profile (see file **$WPEhome/.emacs**):

```
(setq load-path (cons "$WPEeditor/" load-path))
(load "warpshell.el" nil t)
(setq window-min-height 6)
(setq split-height-threshold 6)
(global-set-key "\^X\^L" 'goto-lisp)
```

To install WPE under Gosling's Emacs, we recommend you to add the following lines to your `.emacs_pro` profile:

```
(load "$WPEhome/maclib/process.ml")
(autoload "wsh" "$WPEhome/maclib/warpshell.ml")
(setq split-height-threshhold 6)
(bind-to-key "send-int-signal" "\^C")
(bind-to-key "common-lisp" "\^X\^L")
```

The variable SPLIT-HEIGHT-THRESHHOLD determines the minimal size of lines of an Emacs buffer. Note the incorrect spelling of *threshhold* in Gosling's Emacs. We recommend to use 6, because WPE uses screen buffers extensively. The Mock Lisp function SEND-INT-SIGNAL is bound to CTL C to send an interrupt signal to Emacs. This is useful for interrupting the Warp shell in case it is doing something you don't want it to do. The last line binds the key CTL X CTL L to position the cursor at the end of the Warp shell buffer. This is useful if you are typing to an other editor buffer and you want to return to the Warp shell buffer.

Note, that Gosling Emacs spells *threshold* incorrectly. As in GNU Emacs, we recommend to bind the key CTL X CTL L to position the cursor at the end of the Warp shell buffer, which is implemented by the Lisp function GOTO-LISP.

If you are using GNU Emacs, but prefer to use the key bindings from Gosling's Emacs, add the following lines to your `.emacs`:

```
(load "gosmacs" nil t)
(set-gosmacs-bindings)
```

WPE can be executed under SUNTOOLS or the X window manager. The display of images on Sun screens is

---

[4]Richard Stallman, GNU Emacs Manual, 4th edition, Free Software Foundation, Cambridge, Mass., February 1986.

[5]James Gosling, Emacs - Screen Editor, Version 264, UniPress Software, Inc., Edison, N.J., 1983.

supported only under X. If you are familiar with Gosling's Emacs and SUNTOOLS and would like to use GNU Emacs and X, execute the shell script `$WPEbin/get-gnu-and-x-defaults`. It will copy the files `.emacs`, `.uwmrc`, `.xtools` and `.Xdefaults` into your home directory. The keybindings are chosen to make X (almost) look like SUNTOOLS and GNU Emacs (almost) look like Gosling's Emacs. Add the lines

```
setenv DISPLAY unix:0
xinit xtools
```

to your `.login` file to invoke X at login time.

1.2 How to run the Warp Programming Environment

The Warp Programming Environment runs on top of Common Lisp under UNIX 4.2 on a SUN-3. If you are using the CMU implementation, you have to have an account on a SUN-3 which has access to the file server **kiwi**. The Warp Programming Environment can be started from any terminal that is connected to such a SUN. The startup time is between 10-20 seconds. To set up the correct path names and environment variables, add the following line to your `.login`:

```
source $WPEhome/USE_CURRENT
```

where **$WPEhome** must be replaced by the system dependent path name prefix[6].

To start WPE from the C shell without an editor, type

```
$WPEbin/wsh
```

This command loads the WPE core image and enters the top level of the Common Lisp interpreter. When the prompt of the Lisp interpreter appears, type `(warpshell)` to enter the Warp shell.

To start WPE from the C shell using GNU Emacs, type

```
$WPEbin/wpeg
```

To start WPE from the C shell using Gosling's Emacs, type

```
$WPEbin/wpe
```

These commands automatically fire up Emacs with a screen buffer for the Common Lisp interpreter, load the Warp Programming Environment and start up the Warp shell[7]. While WPE is being loaded, the mode line

```
Warp Programming Environment: Warp Shell (Common Lisp)
```

is displayed if you are using Gosling's Emacs. With GNU Emacs, the only difference in the mode line is that `Common Lisp` is replaced by `Inferior Lisp`. Once loading is finished, the mode line display shows the version number postfixed by the letter C (which stands for core image), the Warp host and the user type:

```
Warp Programming Environment 2.6C: Warp shell (Host: warpb User:developer)
```

1.3 System Configuration

Figure 1-1 shows the system configuration of the Warp Programming Environment used at Carnegie-Mellon University. The system consists of a set of workstations connected to each other via an Ethernet. The majority of the workstations are diskless. Workstations with attached disks are called *file servers*. Each workstation, a SUN-3,

---

[6] If you want to use the experimental version, add the line `source $WPEhome/USE_EXP`.

[7] The **wpe** command is an alias for 'exec emacs -ewsh'. The **wpeg** command is an alias for 'setenv EIPCCHAN E_IPC_$$; exec gemacs -e wsh'. If you are already running Emacs and want to invoke the Warp shell from within Emacs, call the Lisp function **pcwsh** which is defined in the editor customization files **warpshell.ml** and **warpshell.el**, respectively.

can run one or more Warp shells or standalone[8] programs. A SUN-3 workstation called *Warp host* (also called master processor in other papers) is physically connected via a bus repeater to the VME bus of the *external host*. The external host consists of two cluster processors CP1 and CP2 with associated memories, a support processor, graphics devices for I/O and two switches SW1 and SW2, all connected to the VME bus[9]. The switches allow the clusters to send and receive data to and from the Warp array (Cell 1...Cell 10) through the interface unit (IU) of the Warp machine. Thus the Warp host is the intermediator between the workstations and the Warp machine itself. Figure 1-1 shows two Warp hosts connected to two Warp machines.

The Warp host runs three kinds of servers: the *Warp server*, the *WPE server* and the *user servers*. The Warp server manages the use of the Warp machine connected to the Warp host. It maintains the queue of users having requested the Warp machine. The WPE server accepts remote requests to open connections to the Warp machine. For each Warp shell and for each standalone program not running on the Warp host, the WPE server creates a "shadow" process called the *user server* on the Warp host side. Each user server provides the functionality of the Warp monitor described in section 3. The main purpose of the user server is to allocate and manage memory: Every time the Warp server allocates the Warp machine to a particular user, the user memory is copied onto the memories of the cluster processors (C1M1 and C2M2 in figure 1-1) and every time a user unlocks the Warp machine, the memory of the cluster processors is copied back into the memory of the corresponding user server. This makes it possible to multiplex the Warp machine between multiple users and maintain user specific state information across several locks/unlocks of the Warp machine.

In the following we use the terms *remote* and *local* to characterize the location of a program in relation to the Warp host: Programs running on a Warp host are in local mode, programs running on other workstations are in remote mode. Only remote programs have user servers associated with them and the communication between remote programs and user servers is via the TCP-IP protocol. Local programs don't need user servers: the Warp monitor is directly linked into their code.

1.4 Software Components

Figure 1-2 shows the major software components of the Warp Programming Environment. The W2 compiler, the W2 debugger and the W2 simulator support the development of Warp programs. The editor, window manager and generalized image library are used for the preparation of programs and for the display of data structures and images. The Warp monitor and the Warp shell support the execution of Warp programs. Communication between the components is via the WPE database. Programs for the Warp array are written in W2 and compiled by the W2 compiler. The result of the compilation is an abstract syntax tree which is accessible by the other components of the Warp Programming Environment. For example, the W2 debugger inspects the syntax tree when it searches for the value of a variable; the Warp shell inspects the syntax tree when it matches the actual parameters of a Warp program call with the formal parameters of the program.

The Warp shell provides the basic functionality of UNIX shells, such as the C-shell, as well as commands to compile and execute programs on the Warp array. It maintains a set of environment variables such as SOURCEFILE (the name of the current W2 program), WARP (the name of the current Warp machine) and BREAKPOINTS (the set of currently defined breakpoints). These environment variables can be inspected and assigned new values with Warp shell commands. The semantics of assignment is different for each environment variable. For example, assigning a

[8] In this manual the term *standalone program* denotes an application program that does not use the Warp shell, but is calling Warp monitor functions directly.

[9] The support processor is not shown in figure 1-1. For a more detailed view of the Warp machine architecture see Annaratone et. al. "Architecture of Warp", Compcon Spring 1987, IEEE Computer Society, 1987.

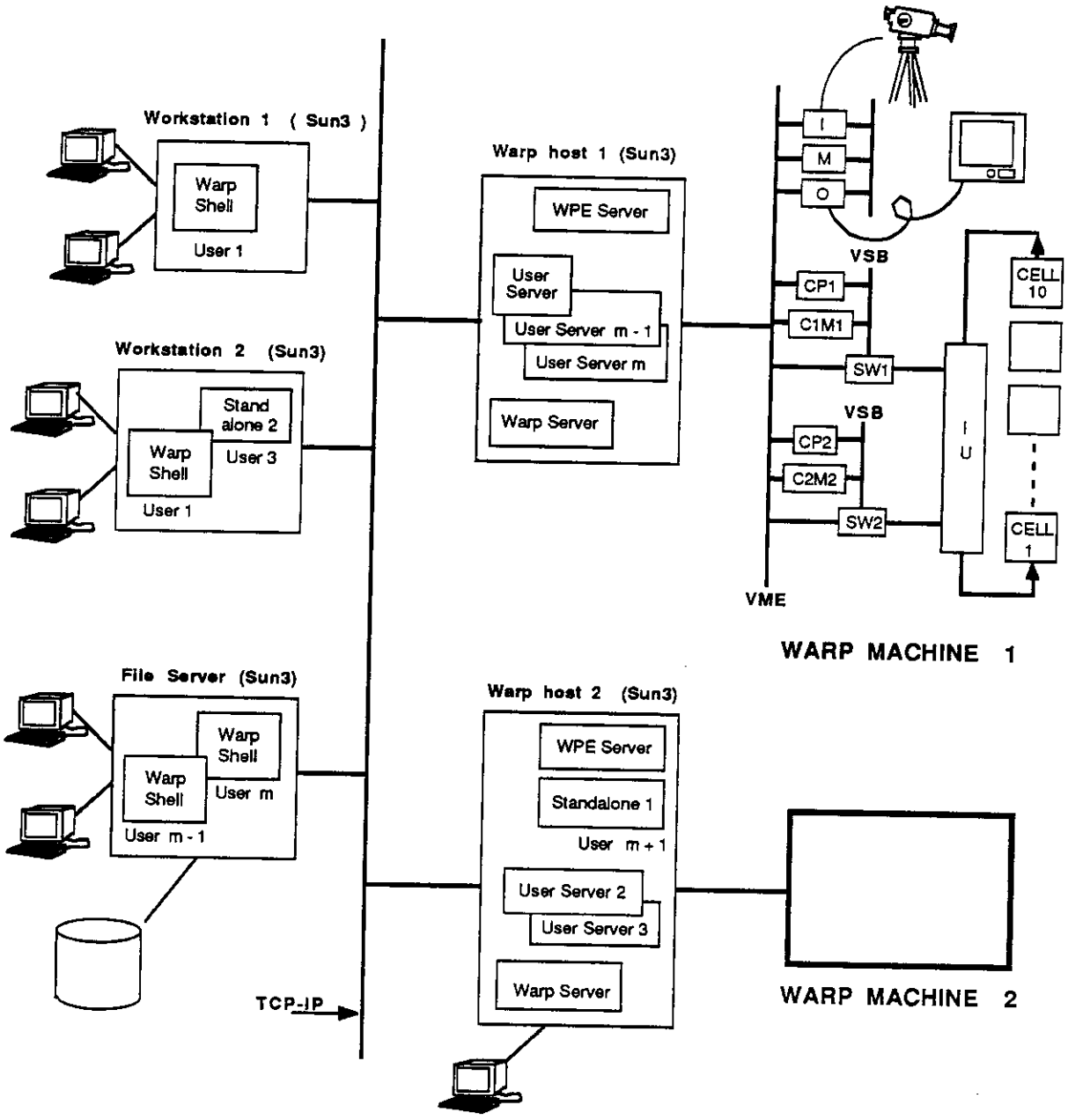

Figure 1-1: System Configuration of the Warp Programming Environment

string "foo.w2" to SOURCEFILE implies the compilation of the W2 program "foo", whereas assigning a value to HOST changes the new Warp host which implies the connection to a different Warp array.

The Warp shell is programmed in Common Lisp whose garbage collector makes it impossible to achieve predictable response times at the shell level. This is tolerable when developing Warp programs, but not when real-time behavior is needed. The Warp monitor supports applications which do not need the full functionality of the Warp shell and must run as fast as possible. In this case programs can run in *standalone* mode by calling the Warp monitor functions explicitly.

Standalone mode is supported for remote and local execution. In remote mode - **Standalone 2** in figure 1-1 -

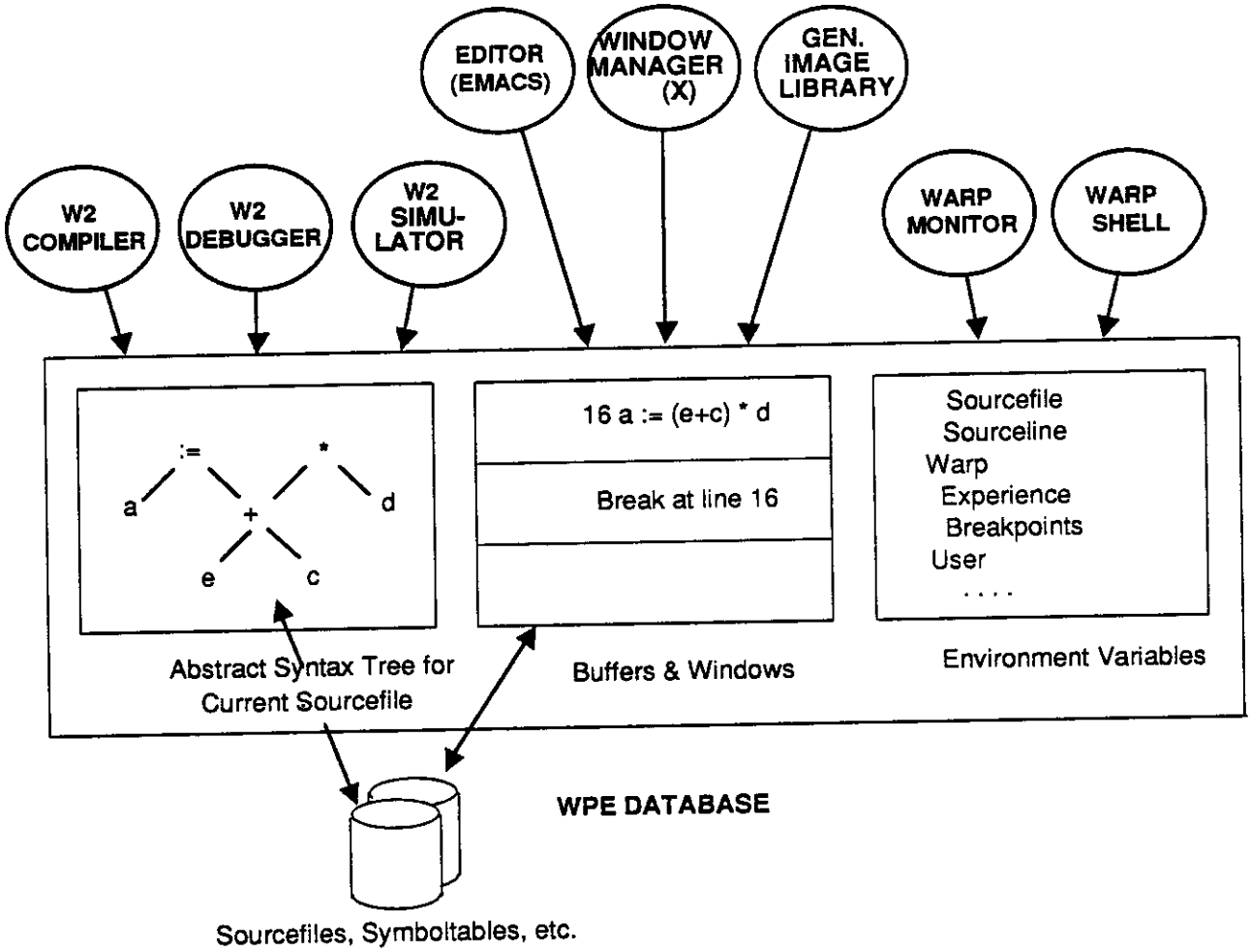

Figure 1-2: Software Components of the Warp Programming Environment

the application is using remote procedure calls implemented with the TCP-IP protocol. In the local mode - **Standalone 1** in figure 1-1 - application programs are running on the Warp host and the Warp monitor functions are executed as direct procedure calls. The local standalone mode is the mode with the lowest overhead and is the preferred mode of execution when time is critical. However, the Warp monitor is implemented such that there is no difference between the core image of an application running in remote mode or in local mode. Thus application programmers can compile, link and test their applications in the familiar environment of their personal workstation before they download them to the Warp host for local execution. Application programs in standalone mode can be written in any language as long as the language implementation supports the call of external C routines (the Warp monitor is written in C).

WPE can be executed inside a window of the X window manager. In this case, other X windows can be used for the display of images. The display of these images is done via the generalized image library. The generalized image library is current being developed at Carnegie-Mellon University, providing device independent access to a variety of image devices and disk formats[10].

[10]Leonard Hamey, *"A User's Guide to the Generalized Image Library"* and Leonard Hamey, Harry Printz, Doug Reece, Steve Shafer, *"A Programmers's Guide to the Generalized Image Library"*, Department of Computer Science, Carnegie-Mellon University.

The Warp shell can be executed inside one of two editors: Gosling's Emacs or GNU Emacs. The communication between WPE and these editors is as follows: Gosling's Emacs sets up a UNIX socket that accepts commands from an external process (in our case WPE) and evaluates them as Mock Lisp commands. The name of the socket is automatically put into the UNIX environment variable EIPCCHAN. The same communication mechanism is simulated in GNU Emacs as follows: The name of the socket is manually put into the UNIX environment variable EIPCCHAN and a subprocess is started by `start-share-filter`[11] which accepts commands from WPE and prints them to its standard output which is fed into an output filter[12] which evaluates them as a GNU Emacs Lisp command.

Both, GNU Emacs and Gosling's Emacs, support parallel editing, that is, the user can continue editing while the Warp shell is executing a command. They also support features such as compiler error message positioning (CTL X CTL N), display of Warp shell variables in buffers, script files, etc. It is also possible to use the built-in Common Lisp editor (**ed**), but in that case it is not possible to do parallel editing. Finally, the Warp shell can be executed without an editor, but in this case none of the editor features are available.

1.5 Organization of the Manual

Chapter 2 deals with the Warp shell. Section 2.2 contains all the commands offered by the Warp shell. Section 2.3 is intended for users who want to do simple customizations of the Warp shell. Section 2.4 to Section 2.7 are for the advanced user who wants to change the functionality of the Warp shell. In Section 2.8 we walk through an example session with the Warp shell and explain the use of the commands.

Chapter 3 describes the Warp monitor. Section 3.3 to Section 3.11 list the functions currently implemented in the Warp monitor. In Section 3.12 we show how to write and execute standalone programs using the Warp monitor. All program examples have been tested.

Chapter 4 contains a list of known bugs and some error messages that might be printed while running the Warp shell. It also explains how to monitor the message traffic between the servers and the Warp shell.

Appendix I summarizes the currently implemented Warp shell commands and Appendix II lists the Warp monitor commands.

1.6 Acknowledgments

The Warp shell is based on the Lisp shell developed by Dario Giuse. The W2 compiler was developed by the compiler group consisting of Chang-Hsin Chang, Robert Cohn, Thomas Gross, Monica Lam, Peter Lieu, Abu Noaman and David Yam. The W2 simulator was implemented by Angelika Zobel. The Warp monitor was implemented by Michael Browne. Ed Clune suggested several extensions to the Warp monitor. Parts of the Warp monitor are based on an earlier runtime environment called WARPCI developed by Francois Bitz and Jon Webb. The generalized image library was developed by Leonard Hamey. Marco Annaratone, Robert Cohn, Harry Printz and Leonard Hamey were the first users and made valuable suggestions.

[11] see `$WPEeditor/warpshell.el`

[12] An output filter is a GMU Emacs Lisp routine which manipulates the output of a process.

2. The Warp Shell

2.1 Introduction

The Warp shell is based on the Lisp shell currently being developed at Carnegie-Mellon University. Its intended way of use is within a text editor. The implementation described here assumes that the user is running the shell within an Emacs editor. The Warp shell provides an extensive online help facility. Type HELP C to get help on a specific command C. To get help on the shell itself type HELP SHELL. The KEYWORD command can be used to get help on a given topic. It searches the table of all currently defined commands and prints out all commands whose description contains a word that matches the given key. For example, the command KEY FILE currently prints out the following:

```
copy        : Copy one or more files
delete      : Delete files
directory   : List files that match a pattern
find        : Find all occurrences of a file name
load        : load a lisp file
rename      : Rename or move one or more files
search      : Search files for a string
touch       : Change the creation date of a file
type        : Type out a file
clean       : Delete garbage files
grep        : Search files for a string
ll          : List files with size and creation date
```

Command names can be aliased to new names. The alias command can be explored by typing HELP ALIAS:

```
  ALIAS       Define a new alias
  SYNTAX:     alias name [value] [description]
With one argument, show the current alias for <name>.   With two arguments,
set the new alias for <name> to be <value>.    If specified, <description> is
used as a one-line help for <name>.
  EXAMPLES:
   alias ll 'directory -long' 'list file sizes and creation dates'
   alias man 'keyword'
```

For example, if you are familiar with the UNIX C-shell, you can alias KEYWORD to KEY and HELP to MAN, respectively.

2.1.1 Syntax of Shell Commands

The Warp shell provides a standardized mechanism for command lines. A command consists of a sequence of words separated by blanks and command options. An option is assumed to start with a minus sign. Options can be either switches or (name,value) pairs. A switch is an option that does not take any argument. A (name,value) pair is an option keyword followed by a value. For each (name, value) pair there is a default value which usually can be found with the HELP command. Commands and options can be uniquely abbreviated, since the shell performs name completion of commands as well as options.

Command line input is not bound to one text line, but can span across several lines of input. If the input cursor is at the end of the last command line, the command reader takes the whole piece of text between the last prompt and the end of the line and passes it to the shell for evaluation. If the input cursor is not on the last command line, but somewhere on a command line earlier in the editor buffer the whole command line that surrounds the input cursor is inserted at the end of the buffer (after the last prompt), and immediately executed.

After the command text has been extracted by the command reader, it is passed to the Warp shell for evaluation. If it starts with an open parenthesis it is treated as a Common Lisp expression and the Warp shell passes it to the Lisp interpreter. Otherwise it is treated as a Warp shell command. This means that the user can type any

combination of shell and Common Lisp expressions to the Warp shell. This makes it possible to write very powerful and flexible shell scripts.

2.1.2 Warp Shell Objects

A Warp shell object is a type or a variable. The Warp shell provides predefined variables called environment variables and user defined variables. Currently the following environment variables are defined:

CELLS — The list of Warp cells to which Warp shell commands apply.

DISPLAY — The display device to be used when displaying data or images. Possible values are: The name of a generalized image as specified in the GIL manual[13] such as "net:warpm:cube", "xwindow", etc, or one of the strings "buffer", "window1", "window2" and "monitor". The default value is "buffer". The value "buffer" binds the display to an Emacs screen buffer, "monitor" binds the display to the Datacube display attached to the Warp machine. and "window1" and "window2" are X windows created by the generalized image library. These windows can be reused when displaying images. Using the generalized image name "xwindow" will open a new X window every time an image is displayed. X windows are created interactively with the mouse as described in the W2-SHOW command in Section 2.2.25, page 25.

DEMO — This environment variable determines how Warp shell commands in command files are executed. Possible values are : "on" or "off". The default value is "off". When DEMO is "on", then Warp shell commands are executed in single step mode, that is, after each command the user is prompted for an action to proceed with the execution of the command file or abort it.

EDITOR — Indicates whether the Warp shell is executed inside an Emacs editor or not. Possible values are: "Emacs" (for Gosling's Emacs), "Gnu-Emacs" and "off". The default value is determined by the way WPE is invoked. If EDITOR is not "off", several Emacs features will be used by the Warp shell:

- When compiling a W2 program, a separate window called `W2 Error Messages` is used for compiler diagnostics (In Gosling's Emacs the name of the screen buffer is error-log, in GNU Emacs it is called *compilation*). This window can be scanned for error messages using the ^X ^N command. Notice that the W2 compiler compiles a preprocessed version of your source program. Thus the error messages apply to the preprocessed version and not to the original file!
- When a breakpoint is encountered the source file will be opened in a separate window and the source line of the current breakpoint is shown at the top of that window. The breakpoint is shown in the file of the preprocessed program, not in the original source program.
- When command files are executed, the current command is displayed in a separate window. When in expert mode (see below), the current command will be shown in Emacs' mini command buffer. When in beginner mode (see below), the command file will be shown in a separate window and the current command is always displayed at the top of the window.

EXPERIENCE — Experience level of user. Possible values are "beginner" and "expert". The default value is "beginner". If you are a beginner, the command NOISY ON is automatically executed at startup time, which means the response to Warp shell commands is shown.

USER — User type. Possible values are "user", "developer" and "implementor". The default value is "developer". If you are a user, you can use all the basic shell commands and the Warp shell commands for the allocation of variables, for the execution of W2 programs and for the inspection of the results. The W2 programs themselves are treated as "black boxes". Thus, compilation and debugging of W2 programs is not possible in user mode. If you are a developer, you have all the rights of a user and the W2 compiler and W2 debugger are available.

---

[13] Leonard Hamey, *"A User's Guide to the Generalized Image Library"* Department of Computer Science, Carnegie-Mellon University, 14 April 1987.

In developer mode, the Warp shell is treated as a "black box". Thus, changing the functionality of the Warp shell is not possible in developer mode. If you are an implementor, you have all the rights of a developer and you can change the Warp shell. For example, you are able to load Common Lisp files into the Warp shell. And you are able to invoke the Common Lisp debugger to debug the Warp shell. You will also get a set of error messages that are not shown in user or developer mode. For example, the Warp shell will automatically deallocate all Warp shell variables in the user server process whenever there is a problem with the server. The implementor is notified of this event but not the other user types. The user type determines the order in which the Warp shell tries to interpret a command. For the user types "user" and "developer", any command typed to the Warp shell is first tried as a Warp shell command and if there is no such command it is tried as a Unix command. For user type "implementor", the Warp shell first tries to evaluate the command as a Lisp variable, and if that is not successful, it is tried as a Warp shell command and then as a Unix command.

MODULE — The name of the module of the current W2 program. The default value is NIL.

SOURCEFILE — The name of the source of the current W2 program. The default value is NIL.

SOURCEFILEDIRECTORY — The name of the directory of the current W2 program. The default value is NIL.

HOST — The Warp host. Possible values here at CMU are: "warpb", "warp8" and "warpm". The default value is "warp8".

SERVERCOMMENT — Comment displayed in Warp server queue entry while requesting or using the Warp machine. Strings with blanks have to be enclosed by single quotes. The default value is "Running WPE". When debugging a program foo.w2, the default value is changed to "debugging foo.w2".

FUNCTION — The name of the current function of the current W2 program. The default value is NIL.

PROMPT — The prompt of the Warp shell. Strings with blanks have to be enclosed by single quotes. The default value is "%".

MINSOURCELINE — The smallest source line number in the current function. The default value is 0.

MAXSOURCELINE — The largest source line number in the current function. The default value is 9999999999.

WARP — The target machine. Possible values and the default value are installation dependent. Most Warp shell commands can be used for any value of WARP, but their interpretation depends on the actual value. For example, If WARP is set to the name of a Warp machine, the W2-COMPILE command generates C code for the cluster processors and W1 microcode for the Warp array. If WARP is set to "simulator", it generates C++ code. Similarly, W2-EXECUTE executes either C++ object code or Warp object code[14]. Naturally, the Warp locking commands do not work if WARP is set to "simulator".

The current value of the environment variables can be determined with the command W2-SHOW -ENVIRONMENT. It is possible to change the value of environment variables with the W2-SET command. The semantics of this command is different for each environment variable. For example, the command W2-SET -SOURCEFILE TEMP.W2 sets the current source file to `temp.w2` which might imply a call to the W2 compiler to compile the program! The command W2-SET -HOST WARPB changes the Warp host to "warpb" and in addition it changes the environment variable WARP to the name of the Warp machine associated with "warpb".

In addition to environment variables, the Warp shell provides the ability to define user-defined objects. Warp shell types and variables can be defined with the W2-TYPE and W2-VAR commands, respectively. Warp shell variables can be initialized with W2-SET, displayed with W2-SHOW and modified with W2-EDIT. Individual Warp shell variables are displayed in editor buffers. If the variable is allocated in cluster memory, the contents is copied into a file in the **/tmp/** file structure before it is displayed[15]. The naming scheme for these files is of the form

[14]Execution of C++ code is not yet implemented.

[15]NOTE: If the file structure is full, you will get an error message when you are trying to display or edit a Warp shell variable.

/tmp/var.name.PID, where name is the name of the variable and PID is the UNIX process id of the Lisp process. If a Warp shell variable is edited that has not yet been initialized, a temporary empty file is created and visited. The naming scheme for these files is of the form **/tmp/var.$name$.PID**. These temporary files are deleted when the corresponding Warp shell variables are deleted with the W2-DELETE command.

2.1.3 Customizing the Warp Shell

After the Warp Programming Environment has been invoked, the Warp shell looks at a command file **warpshellinit.cmd**. This file is looked up in three places until it is found: First in the current directory, then in the user's home directory, and finally in the directory **$WPEhome/**. If you want to do your own customizations, copy the file **$WPEhome/warpshellinit.slisp** into your current directory or home-directory and modify it. The command file can contain any sequence of Warp shell commands as well as Common Lisp commands (see @ command).

2.2 Warp Shell Commands

The Warp shell offers two kinds of commands: First, commands such as delete, copy, remove, etc. These commands are not described in this manual, because it is assumed that the reader is already familiar with them. Type HELP COMMANDS to explore them. Second, commands to compile, debug and execute w2 programs on the Warp. These commands are described in the following.

Depending on the user type, the Warp shell looks up commands in a certain order. If you are user type "user" or "developer", any command typed to the Warp shell is first tried as a Warp shell command. If there is no such command, it is interpreted as a Unix shell command. If you are an "implementor", the Warp shell tries to evaluate the typed command as a Lisp variable. If that is not successful, it is successively tried as a Warp shell and a Unix shell command.

UNIX shell commands are executed by invoking the shell defined in the UNIX environment variable SHELL[16].

A summary of all the Warp shell commands is contained in the Appendix I.

2.2.1 Interrupting Warp shell Commands

You can abort any Warp shell command currently being executed and return back to the Warp shell command interpreter. If you are using WPE inside GNU emacs, type CTL C twice to abort Warp shell commands. This is a special feature of GNU Emacs: The Warp shell runs in Shell mode, which defines several special keys attached to the `C-c` prefix[17]. In particular, `C-c C-c` is bound to `interrupt-shell-subjob`, which sends an interrupt character to the shell. Thus, if you type CTL C once, it will appear as **C-c-** in the message window. Now type another CTL C: You will see **C-c C-c** and the interrupt is generated. When the CTL C is acknowledged by the Warp shell, you will see the message

```
>>Interrupt: Keyboard

SYSCALL:
:A Abort to Lisp Top Level
:C Resume interrupted instructions
```

[16]If the environment variable SHELL is set to the C-shell, the file .cshrc is visited *every* time a UNIX command is executed. If your .cshrc file redefines the shell prompt, the prompt will be echoed by the Warp shell. The approved way to avoid echoing of the prompt is

```
if ($?prompt) set prompt = <promptstring>
```

[17]See Section *Shell Mode* in the GNU Emacs manual for more details.

If you now type :a, the Warp shell returns back to its top level. If you type :c, the Warp shell will try to continue with what it was doing when the interrupt occurred.

2.2.2 @ Command

The @ command executes a Warp shell command file. The Warp shell permits the execution of command files that can be nested to an arbitrary depth. Warp shell command files can contain any sequence of Warp shell commands as well as Common Lisp code. An example of an initial command file is:

```
WPEVersion
(format t "Type help for help~%~%")
w2-set -host warpm
;@$WPEhome/test/pipe/pipe.cmd
```

This command file prints out the WPE version and the string `Type help for help` followed by two carriage returns. The Warp host is set to "warpm". The last line calls a command file located in **$WPEhome/test/pipe/** but it is commented out (using the Common Lisp comment ";").

Common Lisp code inside a Warp shell command file that extends over more than one line has to be written as follows: The last character of each Lisp line except for the last one has to be terminated by a semicolon (;). Warp shell commands can be called within a Common Lisp function but must be passed as strings to the Warp shell function SHELL::SH. For example, the following three Common Lisp lines check whether WPE is running inside GNU Emacs. If yes, the Warp shell command W2-SET -SHOWCOMMANDFILE ON is executed. If not, W2-SET -SHOWCOMMANDFILE OFF is called:

```
(when (string-equal  shell::*editor* "Gnu-Emacs");
        (shell::sh "w2-set -showcommandfile on");
        (shell::sh "w2-set -showcommandfile off"))
```

During their execution command files can be shown in a separate Emacs buffer. If the command W2-SET -SHOWCOMMANDFILE ON has been issued, the currently executed command of the command file is displayed. The display mode depends on the value of the environment variable EXPERIENCE. If it is set to "beginner", the whole command file is shown in a separate screen buffer and the currently executed command is always positioned at the top of the buffer. If EXPERIENCE is "expert", the command file is not shown but the currently executed command is displayed at the bottom of all the editor windows.

NOTE: If the current directory is changed within a command file, it is restored to the original value when the Warp shell returns from the command file.

2.2.3 PAUSE

```
PAUSE       Pause the Warp shell
SYNTAX:     Pause [-silent]
OPTIONS:
 -silent: Don't explicitly prompt user for carriage return
EXAMPLES:
 Pause
 Pause -silent
```

PAUSE pauses the Warp shell until the user types a response. The three response options are: Continue, Skip or Abort. The PAUSE command is issued automatically after each command when in demo mode (see W2-SET -DEMO).

2.2.4 QUIT

```
QUIT        Quit the shell
SYNTAX:     quit [-noconfirm -save F]
OPTIONS:
 -noconfirm: Don't ask for confirmation [Switch (Default if expert)]
 -save: Write current core image in file F [Default file name: wpecoreimage]
EXAMPLES:
 quit
 quit -save /usr/bob/bin/wpe
```

QUIT deletes all temporary files created during the interactive session and exits the Warp Programming Environment. If you are an implementor and the Lisp variable ***load-core-image*** was set to t when a customized Warp Programming Environment was loaded (See Sections 2.3 and 2.4.5), the core image of WPE can be saved in a file specified in the '-save'n option.

2.2.5 RECORD

```
RECORD      Record the Warp monitor calls
SYNTAX:     Record [on|off]  -file F
OPTIONS:
 -file: Filename for recording the commands [No default]
EXAMPLES:
 record on -file /usr/bob/wpe/record.c
 record off
```

The idea behind this command is to support the user who would like to write standalone programs, but is not yet familiar with the necessary calling sequences. The RECORD makes it possible to explore the use of the Warp monitor functions (see Chapter 3) inside the Warp shell. RECORD ON -FILE F creates a file F and writes all Warp monitor calls into the file until a RECORD OFF is issued which turns the recording off and closes the file.

2.2.6 WPEVERSION

```
WPEVERSION  Print the version number of WPE
SYNTAX:     WPEVersion
SEE ALSO:   version
OPTIONS:
 -all: Print the version numbers of all the components of WPE.
EXAMPLES:
 wpeversion -all
```

WPEVERSION prints the current version number of the Warp Programming Environment, the date of the release and the date of the last update (which is the time of the last modification of the update file (see Section 2.4.6)). If you have loaded new files into the Warp shell (with the Warp shell LOAD command), the name of each of these files and the time stamp of their last modification are printed. The -all option also prints the version numbers of the W2 compiler, W2 debugger, Warp monitor, Warp server, Lisp shell, generalized image library and Gnu Emacs.

2.2.7 W2-BREAK

```
W2-BREAK    Set a source line break point
SYNTAX:     w2-break sourceline filename
 OPTIONS:
 -action: Action to be executed when breakpoint is encountered.  [Default: ("w2-halt")]
 -cells:  list of cells to which the breakpoint should be added [default: all cells].
 -condition: Predicate to be evaluated when breakpoint is encountered.   [Default: T]
 -enabled: Specifies whether breakpoint is initially enabled or disabled.[Default: T]
 -file: source file name [default: CurrentSourceFile].
 -function: Function name [default: CurrentFunction].
 -line: source line number.
 -node: dag node number (can be determined with w2-suggestbreaks) .
 -noselect: require user interaction for node selection when
                                                 breakpoint not unique [default].
```

```
SEE ALSO:   w2-delete  w2-set  w2-show  w2-suggest
EXAMPLES:
 w2-break -line 23 -file test2.w2 -select
 w2-break -line 23 -file sm -cell 1 2 6
 w2-break -node 13 -file test2.w2 -cell 1 2
 w2-break -node 4 -file sm.w2 -action "w2-go"
 w2-break -node 5 -action '("w2-locals" "w2-continue")' -condition '(equal (i j))'
 w2-break -node 13 -function foo -cell 1 2
 w2-break -line 23
```

The W2-BREAK command permits the user to set a breakpoint in the current W2 program. Breakpoints consist of four parts: Status, Cell, Condition and Action part. The status specifies whether a breakpoint is enabled or disabled. The cell part specifies the Warp cells to which the breakpoint applies. The condition part specifies a boolean predicate. If it evaluates to TRUE the action part will be executed. The condition part can be any Common Lisp expression and the action part can be any sequence of Common Lisp and Warp Shell commands. Thus breakpoints are user programmable. The default values for the condition and action part can be changed by setting the environment variables BREAKACTION and BREAKCONDITION, respectively (see W2-SET command). Any breakpoint can be changed dynamically with the W2-EDIT command.

It is possible to mix Lisp and W2 values in breakpoint conditions. W2 values can be retrieved by calling the Lisp function `w2-get` with any argument that is a valid argument to the Warp shell command W2-GET. W2-GET returns W2 values in the internal representation used by Common Lisp. Examples of breakpoint conditions are:

```
(= (W2-GET "i") 1)

(= (W2-GET "-value global") 1.22333)

(and (= (W2-GET "-value i -cell 1") 2
     (= (W2-GET "-value i -cell 2") 3)))

(= (W2-GET "-value array") #(0.1 0.2 0.3))
```

In the following example, the breakpoint condition checks whether the W2 variable `i` has the value 1 every time a breakpoint is encountered. If yes, the breakpoint action W2-HALT will be exeuted. If no, the execution of the W2 program is resumed without any breakpoint action. "Beginner" users are notified every time a breakpoint is encountered, even if it evaluates to false. "Expert" users are only notified of breakpoints evaluating to true.

```
% w2-edit -break 1
Action (Continue, Skip, Abort) [Continue]:
Name: 1    ENABLED              Source Line:  45    in Function:  W2_PIPE
         Condition:     (= (W2-GET "i") 1)
         Action:        ( "w2-halt" )
         Cells:         (1 2 3 4 5 6 7 8 9 10)
         Node:          161    Name:W2_PIPE$A Type:FLOAT Op:$RECEIVE
         W1-Address:    (458)
% w2-set -user beginner
% w2-execute -pipe input output
Copying input parameters into cluster memory...
Downloading microcode for file pipe.w2...
Locking Warp Excalibur...
Warp server queue is empty.
Warp machine is yours:
Starting execution of module...
```

```
Breakpoint 1 encountered in cell 3: Line 45 in "/usrw61/bob/wpe/deb/pipe.w2"
          Condition (= (W2-GET "i") 1) is false. Resuming...
Breakpoint 1 encountered in cell 2: Line 45 in "/usrw61/bob/wpe/deb/pipe.w2"
Executing breakpoint actions...
@break1% w2-halt
break1% w2-set -exp expert
break1% w2-cont
Breakpoint 1 encountered in cell 3: Line 45 in "/usrw61/bob/wpe/deb/pipe.w2"
break1% w2-cont
Breakpoint 1 encountered in cell 4: Line 45 in "/usrw61/bob/wpe/deb/pipe.w2"
break1% w2-cont
...
```

2.2.8 W2-COMPILE

```
W2-COMPILE  Compile a W2 program
SYNTAX:     w2-c [-file] filename [other options (see below)]
OPTIONS:
 -as (default "on"):
  Assemble the output of the compiler (on,off)
 -check: Compile only if W2 source file or W2 compiler version has been
                                                        changed [Switch].

 -clean (default "on"):
  Remove compiler files no longer needed after compilation (on, off).
 -debug: Generate cluster code for debugger (Warp) [Switch]
 -download: Download the microcode onto the Warp array after the compilation [Switch]
 -file: File to be compiled (name of file) [Default value: Current source file]
 -hex (default "off"):
  Produce text output (in hexadecimal) instead of binary output (on,off)
 -host (default "vmx"):
  (vme: Do not use external host for I/O, vmx: Use external host for I/O)
 -keep-error-log: Do not erase Emacs buffer 'error-log' when compilation
                                                        is started. [Switch]

 -m4opt (default ""):
  Pass M4 option to the M4 preprocessor.
 -optimize (default "off"):
  If off, do not generate optimized code.
 -preserve (default "off"):
  Keep the output of the macro expander in FILE-NAME.pw2 (on, off)
 -relink: save files needed for relinking [Switch]
 -remote: Assemble Warp host code on specified remote machine (name of host)
 -silent: Keep the verbosity of the compiler messages to a minimum [Switch]
 -spicy (default 0):
  Charaterizes the size of the W2 program:
      0 indicates a normal program, 10 a very large program (0,...,10)
 -target (default "warp"):
  Target architecture for the W2 simulator (warp, 2D, ...)
 -timing: Print timing information when running the W2 simulator
 -verbose: Do not suppress compiler messages [Switch]
EXAMPLES:
  w2-compile $WPEexamples/w2/pipe.w2
  w2-compile -file pipe.w2 -opt off
  w2-compile pipe -dag 10000 -basic 300 -remote warpb -as on
  w2-compile -file pipe.w2 -preserve on  -download
```

To be able to execute this command, you must be a developer (see W2-SET). The simplest form of the W2-COMPILE command is

```
w2-compile -file foo.w2
```

where foo.w2 is a w2 program. The option key word '-file' and the extension '.w2' are optional. If the Warp shell environment variable WARP is set to the name of a Warp machine, code is generated for the cluster processors and W1 microcode for the Warp array. If it is set to "simulator", C++ code is generated. All W2-COMPILE options are shown above. The '-as on' option assembles the microcode, '-as off' suppresses the assembly. The '-debug' option generates symbolic information needed for the debugger. The '-remote' option is needed when the W2 program is

compiled on a workstation that does not have access rights to the C compiler that translates the cluster code. In this case the '-remote' option can be used to specify a workstation with has access rights to the compiler. If you are compiling large W2 programs, it is possible that certain data structures of the W2 compiler overflow. In this case you can specify larger sizes with the spicy option. For example

```
w2-compile -file foo -spicy 2
```

will compile a medium program (some of the bigger programs in the WEB library, for example `egpr`). The spicy flag will also ask for more space for your lisp system. Currently, the sizes are

```
spicy   Lisp image size
  1              26 MBytes
  2              29 MBytes
  3              32 MBytes
  4              36 MBytes
  5              39 MBytes
```

It is a bad idea to use the spicy flag when there is another LISP/WPE job running on your machine. WPE will ask for the space, but UNIX won't deliver. (You can check the actual sizes while the system is running by using the "ps" command, i.e. "ps ux") You should also keep in mind that the assembler & post-processors need some space of their own. If you have 35 MBytes of swap space and run with -spicy 3, your post-processing jobs will die, and you get the message "error in postprocessing". Consider if you really need such a big WPE job, or use a machine with more swap space.

If the first source line of a w2 program contains a comment of the form

```
/* configure O1 V1 O2 V2 .....On Vn  */
```

where `Oi` is a w2-compile option and *Vi* is its value, then the program is compiled with these options. Options added to the W2-COMPILE command line overwrite options in the source program. For example, if the first line in foo.w2 contains

```
/* configure -spicy 2 */
```

and you type

```
% w2-compile foo.w2 -spicy 3
```

the w2 compiler is reconfigured with spicy option 3.

When compiling a W2 program, a separate Emacs window called `W2 Error Messages` is used for compiler diagnostics (In Gosling's Emacs the name of the screen buffer is **error-log**, in GNU Emacs it is ***compilation***). This window can be scanned for error messages using the ^X ^N command. If the W2 compiler encounters an internal error when in developer or user mode, the following message is printed out:

```
INTERNAL ERROR in W2 compiler:
   Keyboard interrupt or unknown Compiler Bug.
Please send mail with details to system maintainer.
```

The W2-COMPILE command is also available as a Unix shell command **$WPEbin/w2c**. All the options mentioned above are available. For example,

```
$WPEbin/w2c pipe.w2 -always
```

compiles the program pipe.w2 and returns back to the Unix shell.

2.2.9 W2-CONTINUE

```
W2-CONTINUE        Start or continue current W2 program
SYNTAX:      w2-continue
OPTIONS:
 -in:     Warp shell variables to read from
 -out:    Warp shell variables to write into.
```

Continue the W2-program from current the breakpoint.

2.2.10 W2-DELETE

```
W2-DELETE   Delete a Warp shell object
SYNTAX:     w2-delete option name
OPTIONS:
 -cells: List of cells of breakpoint to be deleted [default: *].
 -type: Delete Warp shell type [no default].
 -variable: Delete Warp shell variable [no default].
 -breaks: Delete breakpoint [no default].
EXAMPLES:
 w2-delete -breaks 5 -cells 1 3
 w2-delete -type pixel
 w2-delete -var *
 w2-delete -breaks *
```

W2-DELETE deletes a Warp shell object such as a type, a variable, or a breakpoint.

2.2.11 W2-DISABLE

```
W2-DISABLE  Disable a breakpoint
SYNTAX:     w2-disable
OPTIONS:
 -breaks: Name of breakpoint.   [* means all breakpoints]
EXAMPLES:
 w2-disable -breaks 5
 w2-disable -breaks 4 5 6
 w2-disable -breaks *
```

W2-DISABLE disables all cells of a currently active breakpoint.

2.2.12 W2-DOWNLOAD

```
W2-DOWNLOAD down load micro code for program onto Warp array
SYNTAX:     w2-download
OPTIONS:
 -all:      download clustercode and  microcode
 -cluster:  download clustercode
 -micro:    download microcode
 -file:     source file name [default: CurrentSourceFile].
EXAMPLES:
 w2-download  -file test2.w2
 w2-download
```

The W2-DOWNLOAD command checks whether the Warp array is allocated. If not, it locks the Warp machine. Then it downloads the micro and/or cluster code for the specified file. The file must have been compiled and linked before. Note that downloading is automatically performed by the W2-COMPILE if the -download option is specified and by the W2-EXECUTE command.

2.2.13 W2-EDIT

```
W2-EDIT     Edit user defined Warp shell variable
SYNTAX:     w2-edit
OPTIONS:
 -break: Name of breakpoint to be edited [No default].
 -var:   Name of Warp shell variable to be edited [No default].
EXAMPLES:
 w2-edit -break 1
 w2-edit -var foo
```

W2-EDIT -VAR V opens an Emacs buffer for the Warp shell variable V and permits you to change its value with any Emacs command. When editing is finished, the buffer has to be saved with a CTL C (There is no other way to save the value!). Note the following distinction: If the Warp shell variable is already allocated in cluster memory, the cluster memory will be updated with the new value, but the file associated with it will not be changed. If the

variable is not yet initialized in cluster memory, the file associated with the variable is changed. If a Warp shell variable is edited that has not yet been initialized, a temporary empty file is created and visited. The naming scheme for these files is explained in section 2.1.2.

W2-EDIT -BREAK permits to change several aspects of a breakpoint: the breakpoint condition, the action to be executed and whether the breakpoint is enabled or disabled. An action can be any Lisp expression or a shell command. It is also possible to write actions consisting of multiple Lisp expressions and Lisp shell commands. In this case they must be enclosed by parentheses and Warp shell commands must be enclosed within double quotes. The following is a transcript of an example session in the Warp shell where the user has set a break point 1 and modifies it twice. First, we want the breakpoint to occur only if i is equal j and halt the Warp machine in this case:

```
% w2-edit -break 1
  Enabled (T or NIL) [nil]: t
  Condition [t]: (= i j)
  Action [w2-continue]: ("w2-halt")
  Cells (List of integers between 1..10) [(1 2 3 4 5 6 7 8 9 10)]: 1 3
```

Then we would like to change this breakpoint such that it occurs unconditionally, prints out the values of the locals of the current function and continues the execution without returning to the Warp shell:

```
% w2-edit -break 1
  Enabled (T or NIL) [T]: t
  Condition [(= i j)]: t
  Action [w2-halt]: ((format t "Locals:~%")
                     "w2-locals"
                     (format t "Continuing execution...")
                     "s w2-continue")
  Cells (List of integers between 1..10) [(1 3)]: 1 2 3 4 5 6
```

Note that any changes to the parts of a breakpoint take place only after the user finishes the whole editing cycle. That is, if the user interrupts the modification of a breakpoint before the editing session is finished, the breakpoint is not changed.

2.2.14 W2-ENABLE

```
W2-ENABLE   Enable a breakpoint
SYNTAX:     w2-enable
OPTIONS:
 -breaks: Name of breakpoint.  [* means all breakpoints]
EXAMPLES:
 w2-enable -breaks 5
 w2-enable -breaks 4 5 6
 w2-enable -breaks *
```

W2-ENABLE enables all cells of a currently active breakpoint specified with the -breaks option.

2.2.15 W2-EXECUTE

```
W2-EXECUTE  Execute w2 module
SYNTAX:     w2-execute
OPTIONS:
 -file: File name of w2 program to be executed (extension .w2 can be omitted)
                                                [Default: Current source file]
 -autolock: Lock and unlock warp machine automatically (on,off) [Default: on]
 -comment: String to be displayed when inspecting the Warp server queue.
                                      [Default: Specified in wpeprofile.slisp].
 -time: Milliseconds elapsed during execution of last W2 program
                        (Write also into file, if file name specified) [Switch]
 -parameters: List of actual parameters for w2 module. Optional
EXAMPLES:
 w2-execute  -file test2.w2 -parameters indata outdata -comment 'Testing test2.w2'
 w2-execute  -f test2.w2 -p indata outdata -time test2.time
 w2-execute
```

The W2-EXECUTE command executes a W2 program specified with the -file option. If the -file option is not specified, the w2 program specified in the last compilation is executed. If there is no such program, an error message is issued. Otherwise w2-execute compares the actual parameter list given in the -parameters option with the formal parameter list of the w2 module specified in the -file option. Actual parameters must have been defined before as Warp shell variables (see W2-VAR command) and must be passed in the same order as defined in the W2 program[18]. Parameter comparison is done by position, that is, the first actual parameter is compared with the first formal parameter, etc. If the types match, the microcode and cluster code of the module is downloaded and the execution is started. It is possible to omit the actual parameter list. In this case the user is asked interactively to provide them. If the -autolock option is on, the Warp machine is automatically locked and unlocked when the W2-EXECUTE command is finished. If the -autolock option is off but the Warp machine is not locked, it is automatically locked and it stays locked after the execution is finished.

2.2.16 W2-GET

```
W2-GET      print a value
SYNTAX:     w2-get
OPTIONS:
 -cells: Cells to inspect (* means all cells) [Default: Cells in which the current
                                                       breakpoint occurred]
 -columns: How many columns to use for printing values  [default: 1].
 -function: Function name (* means all functions) [default: CurrentFunction].
 -var: Variable name [No default].
 -globals: Display the globals of the cell program (external to all functions).
 -locals: Display the locals of the function.
 -mode: Display format (INT FLOAT BINARY OCTAL HEX or *) [Default: W2 Type of variable]
EXAMPLES:
 w2-get  -value temp -cells 1
 w2-get  temp
 w2-get  -locals
 w2-get  -locals -function init -mode *
 w2-get  -globals
 w2-get  temp -mode *
 w2-get  temp -mode hex
```

Print the value of variables. The keyword -value is optional. If it is omitted, the variable name has to be the first argument of the command. If the -locals option is specified, the command prints the values of the specified function. If the function is not allocated, an error message is printed. If the -cells option is specified, the values are shown for the selected cells, otherwise only the values of the cell in which the current breakpoint occurred are are shown. The -mode option permits the user to specify one of several display formats. For example -mode int will display the value as an integer. -mode * will display the value as an integer, float, binary, octal and hexadecimal.

2.2.17 W2-HALT

```
W2-HALT     Halt the Warp Machine
SYNTAX:     w2-Halt
Halt the Warp Machine
```

This command instructs the Warp shell that the Warp machine is halted.

[18] If the user executes a W2 program that has been previously compiled with the W2-COMPILE command, the Warp shell accesses the syntax tree to do the parameter checking. If the syntax tree is not available, the Warp shell tries to access a file whose name is derived from the concatenation of the file root specified in the -file option and the extension `.param`. The `.param` file is a text file that specifies things like the location of the micro code and cluster code, the name of the W2 module and the number and types of the formal parameters.

2.2.18 W2-INIT

```
W2-INIT      Initialize the GIL
SYNTAX:      w2-init
OPTIONS:
 -gil: Init the GIL
SEE ALSO:    w2-load
```

Initializes the generalized image library. If this command has not been issued when the user tries to use the generalized image library, the initialization will be done automatically.

2.2.19 W2-LOAD

```
W2-LOAD      Load a component
SYNTAX:      w2-load [-compiler | -debugger | -warpshell]

OPTIONS:
      -compiler:  Load the w2 compiler.
      -debugger:  Load the w2 debugger.
      -warpshell: Load the Warp shell.
      -startupfile: Name of Lisp file that contains the loading commands.
EXAMPLES:
 w2-load -Warpshell
 w2-load -debugger -compiler
 w2-load -compiler -startupfile /usr/bob/wpe/start.slisp
```

Load a component of the programming environment into the Warp Shell. Current components are the compiler, the debugger and the Warp shell. The command W2-LOAD -COMPILER -STARTUPFILE F can only be used when you have loaded WPE using the **wpel** or **wpegl** command (see Section 2.3), that is, if you don't use the core image version. Loading a compiler into the core image version with another compiler already being loaded is not very useful : old **defvar** declarations won't be overridden.

2.2.20 W2-LOCKWARP

```
W2-lockwarp reserves the Warp machine
SYNTAX:      w2-lockwarp
OPTIONS:
 -queue:    Display the queue of users currently using the Warp or
            requesting it.
 -timeout: Number of minutes before timing out [Default: -1 (forever)].
 -noqueue: Don't list the users who have currently requested Warp [Switch (Default)]
 -comment: String to be displayed when inspecting the Warp server queue.
          (Any string without blanks) [Default: Running_Warp_programming_environment].
 -notifyme: Send a request if somebody wants to use Warp (ON, OFF) [Default: On].
SEE ALSO:    w2-unlock  w2-warpqueue
EXAMPLES:
 w2-lockwarp
```

This command tries to allocate the Warp machine for you. To release Warp, use the shell command W2-UNLOCK. If the machine is already in use by another user a string specified by the -comment option is sent to that user and you are appended at the queue of users waiting for the Warp machine. If the -queue option is used, this queue will be displayed. The head of the queue specifies the current owner of the Warp machine. Once you are at the head of the queue, the message

```
Warp machine is yours:
```

appears on your screen and the Warp machine is locked.

2.2.21 W2-RESET

```
W2-RESET    Reset the W2 debugger, the Warp server or
            the Warp machine to its initial state.
SYNTAX:     w2-Reset
 -verbose: Be verbose when resetting [Switch (no default)]
 -server: Reset the server
 -debugger: Reset the debugger
 -file: Current source file [Default: CurrentSourceFile]
 -Warp: Reset the Warp array
```

Initializes the W2 debugger or the Warp server to its initial state.

The memory descriptors of all allocated Warp shell variables are invalidated. The W2-RESET -DEBUGGER command should be used whenever the debugger is in a dubious state. The command deletes all currently defined breakpoints, and any assumptions about the last compiled w2 file.

The W2-RESET -SERVER starts or restarts the Warp server. This command should be used after it has been determined that the Warp host and file server are working correctly (Use the W2-SANITY command to check the state of the file server, Warp host, Warp server and Warp machine). The W2-RESET -SERVER command might not be successful in restarting the Warp server. For example, if the current Warp server is hung, this command has no effect. In this case, *login on the Warp host* and call **$WARPmisc/reset_server** to reset the server[19]. The `reset_server` program unconditionally kills any existing Warp server even if there are users currently using the Warp machine (because of this reason it should be used with care!). Then it pauses for 60 seconds, tries to restart the Warp server and exits with `"Reset complete"`. However, if you also get the error message `"Cannot bind socket address!"`, the reset was not successful and you have to try again.

The W2-RESET -WARP should only be done after the W2-SANITY command has determined that the Warp machine is in a bad state.

2.2.22 W2-RESTART

```
W2-RESTART  Restart current W2 program
SYNTAX:     w2-restart
```

Restart the current W2 program with the i/o parameters from the last W2-EXECUTE command. Note: W2-RESTART cannot be used if the Warp server has died after the last W2-EXECUTE.

2.2.23 W2-SANITY

```
W2-SANITY   Check the state of Warp host, WPE server, Warp server and Warp array.
SYNTAX:     w2-sanity
Checks the state of the Warp server, WPE server, the Warp host and the Warp array
The state of the Warp machine is checked only if all the other components
seem to be working.  If the Warp host is up, w2-sanity checks its uptime
Warp host. If the -statistics option is specified, it prints several
statistics about the Warp array (see WPE user manual).
OPTIONS:
 -statistics: Print statistics about Warp array
EXAMPLES:
 w2-sanity
```

Because of its distribution over several machines, the Warp Programming Environment is less reliable than a single machine environment. The purpose of the W2-SANITY command is to give the user some help to check the state of the components of the environment.

---

[19]Executing `reset_server` on any other workstation than the Warp host will have no effect. If you are not authorized to login on the Warp host, ask the system maintainer to do the reset for you.

It consists of two parts: First it checks the state of the Warp host, Warp server, WPE server and Warp array. The following responses are possible:

```
---> Internal error in sanity_check.

---> Warp host not up.

---> Warp server not running.

---> WPE server not running.

---> Cannot test Warp array (problem with user server).

---> Warp array is not accessible.

---> Warp host, Warp server, WPE server and Warp array are all accessible.
```

The messages **Problem with user server** usually means that there is not enough memory space available on the Warp host to fork off a user server to do the testing.

If the Warp host is up the command then checks the uptime of the Warp host. For example:

```
Checking uptime of Warp host...
  1:45pm  up 41 mins,  3 users,  load average: 2.73, 2.14, 1.57
```

Finally, if the -statistics option was specified, the command checks several statistics about the Warp array. These statistics characterize the past behavior of the Warp array while running a program called *idle*, a Warp monitor program executed by the Warp server whenever there is no other entry in the Warp server queue. **Idle** repeatedly runs a W2 program which returns known results, compares the actual results with the known values and tabulates the number of failures/successes[20].

The fields have the following meaning:

Total time since Date
: Time of the interval from when reporting started until the moment that the last idle run started or stopped.

Total downtime
: Accumulated down time of the Warp host or Warp array. during the reporting interval.

Total valloc downtime
: Accumulated time where **idle** cannot run because of lack of virtual memory on the Warp host.

Total non-idle time
: Accumulated time of Warp array usage.

System Crashes
: Crashes of the Warp host.

Valloc Failures
: Number of failures during a **valloc** call.

Successes
: Number of successful completions of the **idle** program.

Failures
: Number of completions of the **idle** program.

The following is an example of the behavior of Warp:

```
Checking Warp Array statistics...
Total time since Wed Jul 29 16:51:56 1987 = 380:53:09
Total downtime = 08:17:30
Total valloc downtime = 01:56:55
Total non-idle time = 33:58:13

System Crashes = 20, Valloc Failures = 56
Successes = 3446690, Failures = 0
```

As we can see from this example, the Warp array has been up for more than 380 hours and has been used for almost 34 hours in this period.

[20] **Idle** is located in **$WARPmisc/idle**. For each idle run the Warp server adds information to a file **$WARPmisc/idle.log**.

Note that the W2-SANITY command is only checking whether each of the above components are running. It does not guarantee that they are working correctly. The command issues a remote shell call to the Warp host to look for the Warp server process and applies the following heuristics: If the remote shell call cannot be executed for some reason it returns the message **Internal error in sanity_check**. If the remote shell call times out, it assumes that the Warp host is dead[21]. If it cannot find a process with the name **warpd** running on the Warp host, it assumes the Warp server is dead. If it cannot find a process with the name /**netimaged** it assumes the WPE server is dead. Finally it checks whether the Warp machine is in a bad state.

2.2.24 W2-SET

```
W2-SET      Set an environment or user defined variable
SYNTAX:     w2-set

OPTIONS:
 -binaryfile: Binary file that contains value of Warp shell variable [No default].
 -breakaction (default "w2-halt"):
  Default action for breakpoints.
 -breakcondition (default T):
  Default condition for breakpoints.
 -calls: Call list to be used for other debugger commands [No default].
 -demo: Single step Warp shell commands in command files [Default: off].
 -display (default "buffer"):
  Display device (buffer, monitor, window).
 -echocommandfilecommands: NIL
 -editor: Use text editor facilities (Gnu-Emacs, Emacs, off)
                                          [Default: Depends on invocation].
 -experience: Level of experience (beginner, expert) [Default in wpeprofile].
 -function: Current function [No default]
 -gimage: Generalized image that contains value of Warp shell variable [No default].
 -host: Warp host (warp8, warpb) [Default in wpeprofile].
 -prompt: Prompt string [Default specified in wpeprofile].
 -printarrays: Print arrays in w2-get command (on, off) [Default: off]
 -showcommandfile: Show command file when executing it (on, off)
                                          [Default in wpeprofile]
 -sourcefile: Current sourcefile [Default: File name of last compilation].
 -textfile: Text file  that contains value of Warp shell variable [No default].
 -timeout: Seconds to wait for the execution of Warp programs before timing out
                                                    [Default: 15].
 -user: Type of user (user, developer, implementor) [Default in wpeprofile].
 -value: Value for the user variable [No default].
 -var: User variable to be set [No default].
 -warp: Warp machine (installation dependent) [Default in wpeprofile].
 -servercomment: Comment displayed when waiting in Warp server queue.

EXAMPLES:
 w2-set -calls 0 1 2 3 4 5 6 7 8 9
 w2-set -calls 1
 w2-set -var in -binaryfile "/usr/bob/wpe/input.image"
 w2-set -var out -text "/usr/bob/test/output"
 w2-set -printarrays on
 w2-set -function init
 w2-set -host warp8
 w2-set -editor Emacs
 w2-set -experience beginner
 w2-set -user implementor
```

[21]The remote shell call looks into your .cshrc file before executing the remote command. If there are bugs in your .cshrc file, the W2-SANITY command might tell you that it cannot connect to the Warp host, even if the Warp host is up. In this case, issue the command W2-TRACE -SERVER ON and watch the message traffic. You might get a message like

```
<<< term: undefined variable
```

The problem in this case is that your .cshrc file did not define the C-shell environment variable TERM before it used it.

W2-SET changes the value of user-defined variables defined with W2-VAR and environment variables such as the current target process, the current source file, the active cell list, etc. If -editor is Emacs (=Gosling's Emacs) or Gnu-Emacs, then the Warp shell will assume that it runs inside a text editor and will try to make use of editor facilities such as multiple windows when it is compiling W2 programs, executing command files or displaying breakpoints. Note that setting the Warp host with the -host option automatically sets the Warp machine, and setting the Warp machine with the -Warp option automatically sets the Warp host. For example, W2-SET -HOST WARP8 sets the Warp machine to GE. The `-display` option changes the default display device used for displaying variables with the W2-SHOW command. Display option value "buffer" binds the display to an Emacs buffer, "window" binds the display to an X window and "monitor" binds it to the Datacube display attached to the Warp machine. Note that the default display can be overwritten for any Warp shell type (see W2-TYPE command) and when displaying variables (see W2-SHOW command).

2.2.25 W2-SHOW

```
W2-SHOW      Show environment variables
SYNTAX:      w2-show
 -all: show environment, user defined variables and breakpoints [default].
 -binaryfile: Write Warp shell variable into binary file
 -breaks: show currently defined breakpoints.
 -display (default "buffer"):
  Display to be used (Buffer, <GIL-Name>,  Window1, Window2, Monitor)
 -environment: show environment variables: user type, source file, etc.
 -gimagefile: Write Warp shell variable into a compact GIF format file.
       Filename is checked for extension .gif
 -range: show slices of the specified array-variable(s).
        Specify a pair l:u for each dimension of the array
        (l = lower bound, u = upper bound of slice).
 -textfile: Write Warp shell variable into a text file
 -types: show currently defined Warp shell type T [Default: all types].
 -trywarp (default 5):
  Seconds to wait for Warp to do halftoning for image display on X window.
 -variables: show user defined Warp shell variable V [Default: all variables].
EXAMPLES:
 w2-show -break
 w2-show -variables
 w2-show -var foo -textfile /usr/bob/wpe/foo
 w2-show -var image -gimage /usr/wpe/images/image
 w2-show -types
 w2-show -var brain -display net:warpm:cube
 w2-show -var kung -display window1 -trywarp 10
 w2-show -var foo -range 0:3 0:11
 w2-show
 w2-show -environment
```

W2-SHOW shows environment variables such as the current target process, the current source file, user defined Warp shell types and Warp shell variables and breakpoints. If a file option -binaryfile, -gimage or -textfile is given, the contents of the Warp shell variable is written into a file. Files of type -gimage are created by a space saving compaction algorithm and have the extension .gif. Binary files will be converted into text files for the time of the viewing. The -range option can be used for binary files to display slices of arrays. For example, w2-show -var b -range 0:3 0:13 displays the first 14 elements of the first four rows of (array) variable b.

If no file option is given, the variable will be displayed in the display device specified in the W2-TYPE declaration. A noninitialized Warp shell variable that is associated with a file is initialized in user server memory before it is displayed. Thus the Warp shell always tries to display user server memory instead of file contents. As a side effect, after the Warp shell has detected that the user server has died, for example as a result of a Warp host crash, and a new user server has been created, the W2-SHOW command re-initializes the Warp shell variable in user server memory with the associated file value before it displays it.

The `-display` option can be used to overwrite the display device binding of the type definition. Display option value "buffer" displays the variable in an Emacs buffer. Display option value "window1" or "window2" displays the variable in an X window and "monitor" displays it on the datacube display attached to the Warp machine[22]. <GIL-Name> can be the name of a generalized image as specified in the GIL user manual.

When displaying an image in a X window on a binary screen display, error propagating half-toning is performed on the fly. This permits the binary Sun screen display to be used for grey-level displays. The half-toning algorithm is slow, but results in relatively good looking images. If speed or accuracy is important, the Datacube monitor or a color screen should be used instead. If the display option "window1" or "window2" is used, and the Warp shell variable has been declared of type **array[512 512] of byte**, then halftoning is attempted on Warp using the WEB library routine `display`. This is 20 times faster than the halftoning performed by the GIL and has a much better algorithm. The -trywarp option determines how long the Warp shell tries to lock the Warp when attempting to do the halftoning: If the Warp is not available within the specified time. the halftoning is performed by the GIL[23].

X windows are created interactively with the mouse. If you press the *left* mouse button, the default size indicated by the flashing identification window will be used. The *middle* button permits you to chose the size of the window. Press the button to define the upper left corner of the window, move the cursor to where the opposite corner of the window should be and release the button. Pressing the *right* button produces a window with the upper left corner at the cursor and the lower left corner at the bottom of the screen[24]. At window creation time, the image is transformed by the generalized image library to fill up the whole window. Note that once the window is created, changing the shape of the window does not change the shape of the image.

If the variable is not bound to a file, the value will be displayed in the Warp shell buffer. The contents is copied into a file in the /tmp file structure before it is displayed. The naming scheme for these files is explained in section 2.1.2. The -range option permits the display of slices of arrays. For example,

```
w2-show -var b -range 0:3 0:13
```

displays the first 14 elements of the first 4 rows of variable B.

For each Warp shell type the W2-SHOW -TYPES command displays the name, basetype, bounds (if an array) and the size of bytes. The following shows all predefined types as well as two user defined types `float10` and `intyp`:

```
Warp Shell Types:
    Name            Basetype        Bounds          Size(Bytes)
    integer         integer         ---             4
    float10         float           (10)            40
    byte            byte            ---             1
    unsigned-byte   byte            ---             1
    unsigned char   char            ---             1
    char            char            ---             1
    intyp           integer         (12 12)         576
    float           float           ---             4
    signed-byte     byte            ---             1
    signed char     char            ---             1
```

The display format for all Warp shell variables uses seven columns:

[22]If the -monitor option is used, the Warp is locked before the datacube is opened. This avoids sending a VME reset signal to the external host when somebody uses the Warp while displaying on the datacube, which causes the datacube access to get lost (and sometimes the Warp host to crash).

[23]The halftoning is always performed by the GIL if the display option "xwindow" is used.

[24]For more details on the creation of X windows, see the section *Sizing Windows* in the `X(1)` entry of the UNIX manual.

```
Warp Shell Variables:
    Name          Type        Memory   MDesc  Mode    Init'd Value
    data          float10     clm2     0      text    yes    /usr/bob/wpe/test/poly/data
    copyin        intyp       d/c      -1     binary  no     /usrw62/yam/w2/test/indata
```

The first two columns, Name and Type contain the name of the variable and its Warp shell type. Memory indicates where the variable will be allocated in the cluster memories when the user locks the Warp machine. For example, clm1 means memory 1 in cluster processor 1, and d/c means don't care. The column titled MDesc describes whether the variable has been allocated in the address space of the associated user server. A memory descriptor -1 indicates no allocation, any other integer indicates that the variable has been allocated. Init'd shows whether the user server memory has been initialized or not. If the variable is a file, Value contains the file name and Mode shows whether the associated file is in binary or in text mode.

2.2.26 W2-SUGGESTBREAKS

```
W2-SUGGESTBREAKS suggest possible breakpoints
SYNTAX:      w2-suggestbreaks

OPTIONS:
 -line: source line number range.    (* denotes all source lines of current file)
                                                                    [Default: *]
 -function: function name.    (* denotes all functions) [Default: CurrentFunction]
 -file: source file name [default: CurrentSourceFile].
SEE ALSO:    w2-break
EXAMPLES:
 w2-suggestbreaks  -file test2.w2
 w2-suggestbreaks -line 23 34
 w2-suggestbreaks -line *
 w2-suggest -line 12 36 -function initialize
```

This command suggests possible breakpoints in the specified source line range. Because the w2 compiler is highly optimizing, source lines which look like possible breakpoint candidates may actually not be used. The W2-SUGGESTBREAKS command permits the user to explore the possible breakpoints of the object code on a source line basis.

2.2.27 W2-TRACE

```
W2-TRACE     Turn on or off tracing information for Warp server and editor.
SYNTAX:      W2-Trace
OPTIONS:
 -server: Monitor message traffic between server and Warp shell (on, off) [off]
 -editor: Monitor message traffic between editor and Warp shell (on, off) [off]
SEE ALSO:    w2-lockwarp  w2-unlockwarp w2-sanity
EXAMPLES:
 W2-Trace -server on
 W2-Trace -editor on
 W2-Trace -server off
```

The W2-TRACE -SERVER command is useful whenever the user suspects that there is something wrong with Warp machine or with the user server. For example, it should be used whenever the Warp machine times out. Messages sent to the server are prefixed with ">>>", messages returned from the server are prefixed by "<<<". The format of the error messages is described in Section 4.3. The command W2-TRACE -EDITOR ON is for debugging the interface to the editor. Messages sent to the editor are prefixed with "Editor>>>".

2.2.28 W2-TYPE

```
W2-TYPE      Declare a Warp shell type.
SYNTAX:      w2-type T TD
OPTIONS:
 -display (default "buffer"):
  Display to be used (buffer, window1, window2, monitor, <GI>)
SEE ALSO:    w2-var
EXAMPLES:
 w2-type PIXEL unsigned-byte
 w2-type IMAGE ARRAY[512 512] of pixel
```

The W2-TYPE command defines a type T using the type descriptor TD. If the `-display` option is used, it will be used to select the display when displaying Warp shell variables of type T with the W2-SHOW command (see Section 2.2.25). If T has already been declared the old declaration is no longer available (if NOISY is on, a warning is issued in this case). TD can be composed of other user defined w2-types and any of the types described below:

Type	Size
unsigned-byte	8 bit
signed-byte	8 bit
char	8 bit
integer	32 bit
float	32 bit
array[dim1] of T	(dim1+1) * (size of T)
array[dim1 dim2] of T	(dim1+1) * (dim2+1) * (size of T)

dim1 and dim2 are the upper bounds of arrays starting at lower bound 0. Arrays of higher dimensions than 2 have to be composed by using array of array constructions. For example: 'array[3] of array[2] of array[3] of integer' defines a 3-dimensional array of integers. Dimensions can be separated by a comma or by a space.

2.2.29 W2-UNLOCKWARP

```
W2-unlockwarp  releases the Warp machine
SYNTAX:        w2-Unlockwarp
OPTIONS:
 -killserver: Unlock Warp and also kill the user server.
SEE ALSO:    w2-lock  w2-warpqueue
EXAMPLES:
 w2-unlockwarp
 w2-unlockwarp -killserver
```

This command releases the Warp array locked by a previous W2-LOCKWARP command.

2.2.30 W2-VAR

```
W2-VAR       Declares a Warp shell variable.
SYNTAX:      w2-var -name N -type T -file F -component C
OPTIONS:
 -name: Name of the Warp shell variable
 -type: Type of the Warp shell variable
 -initialize: Initialize variable in cluster memory at declaration time [Switch].
 -input: Initialize variable in input cluster memory at declaration time [Switch].
 -output: Initialize variable in output cluster memory at declaration time [Switch].
 -binaryfile: Initial value is in binary file F
 -gimage: Initial value is in G which is a generalized image
 -textfile: Initial value is in text file F
 -value: Initial value is in lisp variable L
 -noerror: No error if variable is already declared. [Switch]
 -component: Host component where to allocate the variable:
          (sun, clm2,clm3,c2m2,c2m3,slm2,slm3,dk) [default: dk (=don't care)]
SEE ALSO:    w2-type
EXAMPLES:
 w2-var -name foo -type image -binary /usr/bob/test.bin -initialize
 w2-var -name bar -type image -component clm3 -text /usr/bob/test.dat
 w2-var -name bar -type float -value BAR
 w2-var -name result -type image
```

The W2-VAR command declares a variable of type T and binds it to the name N. If N has already been declared, the old declaration is no longer available (A warning is issued if NOISY is on).

T can be a built-in type or a user-defined type (see W2-TYPE command). The -component option specifies the component where the variable will be allocated. Possible components are *"sun", "C1M2", "C1M3", "C2M2", "C2M3", "dk"* (which means don't care).

The W2 compiler generates DMA code by default for all W2 programs compiled without -debug option. To effectively use DMA, *input* Warp shell variables have to be allocated in C1M* (C1M1, C1M2, C1M3) and *output* Warp shell variables in C2M* (C2M1, C2M2, C2M3). The options -input and -output indicate these components. For example, to allocate Warp shell input variable IN and output variable OUT, use

```
w2-var -name IN -type float -input
w2-var -name OUT -type float -output
```

If memory allocation fails in a cluster component specified with the -component option, for example because there is not enough space, a warning is issued and the Warp shell tries to allocate the requested variable in a component selected by the Warp monitor (same as dont't care). This allocation will be successful as long as there is enough memory space available in the memory boards.

The value of the variable can be associated with a binary file, text file or a generalized image file F with the -binaryfile, -textfile or -gimage option, respectively. These options are useful when you want to initialize a Warp shell variable with the contents of a file[25]. If the -gimage option is used, the variable has an initial variable which is contained in the generalized image G. The -value option associates the variable with the value of a Lisp variable[26]. If a Warp shell variable is not associated with a file, then its value is only kept in cluster memory (or in Lisp memory) and is deleted when quitting the Warp shell.

The -initialize option is useful for Warp shell variables used as input parameters for W2 programs. If the -initialize option is used, then the user server memory is initialized at declaration time with the associated value (The Warp shell does not do any type checking during this initialization). Otherwise the initialization is delayed until the input parameter is actually used (for example in a W2-EXECUTE command). Normally, a warning message is issued if the variable is already declared. This is suppressed in expert mode or with the -noerror option.

It is possible to issue W2-VAR commands even if you have not locked the Warp machine. In this case Warp shell variables are allocated in the address space of the user server process. When you finally get access to the Warp machine, the memory associated with the variables is copied into the cluster processor memories. And when the machine is unlocked, the variables are copied back from the cluster memories to the user server's address space.

2.2.31 W2-WARPQUEUE

```
w2-warpqueue Show the users currently owning the Warp machine or waiting for it.
SYNTAX:      w2-warpqueue
SEE ALSO:    w2-lockwarp  w2-unlockwarp
EXAMPLES:
 w2-warpqueue
```

This command shows the current entries in the Warp server queue. If there are no users, the Warp queue is empty. Otherwise the first entry of the queue is the current owner of the Warp machine and the other entries are

[25] In older versions of WPE, it was also possible to associate a file name with a Warp shell variable to store an execution result. This is no longer possible. To write the value of a Warp shell variable into a file, use the W2-SHOW command.

[26] The -value option is not yet implemented.

users who have requested Warp and are waiting for it.

2.3 How to load a Customized Warp Programming Environment

The Warp shell can be customized in various ways, for example by use of the ALIAS command and by changing the default values of environment variables in the initial command file **warpshellinit.cmd**. This section is intended for users who want to customize their shell even further.

The **wpe** and **wpeg** commands start up a core image of the Warp Programming Environment with a stable version of the W2 compiler. If you need to load a customized environment, for example because you are working on an experimental compiler, you have to load the whole environment from scratch. This can be done with the commands

```
$WPEbin/wpegl        (for GNU Emacs)

$WPEbin/wpel         (for Gosling's Emacs)
```

These commands "source" an environment file **warprc** (see Section 2.4.1) and invoke the editor function `loadwsh` instead of `pcwsh`. When WPE is loaded, the version number is printed out, followed by the name of the editor and the full pathname of the directory file **wpe.slisp** (see Section 2.4.4). In the following example WPE is read from **$WPEroot/** inside the GNU Emacs editor:

```
> "Warp Programming Environment 2.6, 08-Jan-88"
> "Gnu-Emacs"
> "$WPEroot/wpe.slisp"
```

The version number in the mode line in the Emacs buffer is postfixed by an L:

```
Warp Programming Environment 2.6L: Warp shell (Host:warpm User:Implementor)
```

Loading the Warp Programming Environment this way takes about 5-15 minutes, depending on the main memory configuration and the job load of your workstation and whether all the components are being loaded. The W2 compiler is not loaded by default. When a user types the W2-COMPILE command the first time, a W2 compiler will be loaded automatically before the command is executed. This compiler is a "stable version" that has successfully compiled a test suite of W2 programs. To load the stable version of the compiler at startup time, the Lisp-variable ***Load-Compiler*** defined in the configuration file (Section 2.4.5) must be set to **t**. To load an experimental compiler at startup time, change the setting of the environment variables `W2root` and `W2compiler` (see Section 2.4.1).

2.4 Customizing the Initial Startup

Six files are initially looked up when loading a customized WPE:

- Environment file (**warprc**)
- Editor profile (**.emacs** for GNU Emacs or **.emacs_pro** for Gosling's Emacs)
- Boot file (**warpshell.el** for GNU Emacs or **warpshell.ml** for Gosling's Emacs)
- Directory file (**wpe.slisp**)
- Configuration file (**wpeprofile.slisp**)
- Update file (**bugfixes.slisp**)

The update file is for temporary bug fixes that have not yet been incorporated into the current release. The configuration file **wpeprofile.slisp** specifies the initial software configuration as well as the default settings of several Warp shell variables. The directory file **wpe.slisp** specifies the location of various directories and files needed when loading WPE. The location of **wpe.slisp** is wired into the boot file **warpshell.ml** or **warpshell.el**,

whose location in turn is determined by your **.emacs** or **.emacs_pro**, respectively. The environment file defines the Unix environment variables which specify the absolute path names of the locations of the WPE components.

The environment file is "sourced" first during a customized startup. The directory file is the first Common Lisp file to be loaded, which loads the configuration file as its first action. Thus the names of all the directories and the initial configuration are known before the rest of the directory file and the other files are executed.

2.4.1 The Environment File: warprc

The environment file defines several Unix environment variables needed by the various users of the Warp Programming Environment. **warprc** is looked up in two places: First in the current directory, then in the user's home directory. If it is found, it is automatically "sourced" before WPE is invoked. If it is not found, it is assumed that the user has already source the warprc file (for example in the **.login** file)).

Figure 2-1 is an example of a subset of the environment variable definitions in a typical a **warprc** file. Note, that all directory names must be terminated with a "/" and that the names are case sensitive.

```
setenv WARPtype      PCW
setenv WARPhost      warpm

setenv WARPbin       /usr/pcwarp/bin/
setenv WARPinclude   /usr/pcwarp/include/
setenv WARPlib       /usr/pcwarp/lib/
setenv WPEhome       /usr/wpe/
setenv WPEroot       /usr/wpe/exp/

setenv WARPserver     /usr/pcwarp/support/server/
setenv WARPmisc       /usr/pcwarp/misc/
setenv WARPexternsrc /usr/pcwarp/host/src/external/
setenv WPEdoc         /usr/wpe/doc/
setenv WPEbin         /usr/wpe/bin/
setenv WPEeditor      /usr/wpe/gnu-ext/
setenv WPEwarpshell   /usr/wpe/exp/warpshell/
setenv WPElispshell   /usr/wpe/exp/lispshell/
setenv W2root         /usr/wpe/exp/compiler/common/
setenv W2compiler     /usr/wpe/exp/compiler/pcw/
setenv W2debugger     /usr/wpe/exp/debugger/
setenv W2simulator    /usr/wpe/exp/simulator/
setenv WPEexamples    /usr/wpe/examples/
setenv WPEdemo        /usr/wpe/demo/
setenv M4include      /usr/wpe/include/
setenv GIL            /usr/vision/experimental/lib/
setenv WindowManager /usr/misc/.X/lib/

set path = ($WARPbin $WPEbin $path)
```

Figure 2-1: Example of a warprc file

If you maintain a private version for any of the WPE components, make your own copy of the **warprc** file and copy it into your working or home directory.

2.4.2 Editor Profiles

If you are using GNU Emacs, your **.emacs** file should contain at least the following lines:

```
(setq load-path (cons "$WPEeditor/" load-path))
(setq window-min-height 6)
(setq split-height-threshold 6)
(global-set-key "\^X\^L" 'run-lisp)
(load "$WPEeditor/warpshell.el" nil t)
```

If you are using Gosling's Emacs, your .emacs_pro file should contain the following lines:

```
(load "$WPEhome/maclib/process.ml")
(setq split-height-threshhold 6)
(autoload "wsh" "$WPEhome/maclib/warpshell.ml")
(autoload "common-lisp-mode" "$WPEhome/maclib/common-lisp.ml")
(bind-to-key "common-lisp" "\^X\^L")
```

Note that **common-lisp-mode** rebinds the CTL Z key. To make the **common-lisp-mode** available when visiting files with the extension ".slisp" or ".lisp", add the MLisp lines

```
(auto-execute "common-lisp-mode" "*.slisp")
(auto-execute "common-lisp-mode" "*.lisp")
```

to your **.emacs_pro**.

2.4.3 The Boot File: warpshell

If you are using GNU Emacs, the default location of the boot file is **$WPEeditor/warpshell.el**. If you are using Gosling's Emacs, the default location of the boot file is **$WPEhome/maclib/warpshell.ml**.

The boot file specifies the location of the directory file. This is done in the Lisp function `loadwsh` by setting the variable `*LoadingFromDirectory*`:

```
(setf *LoadingFromDirectory* (concat (getenv "WPEroot" "wpe.slisp")))
```

If you want to load the WPE version from /usr/foo/bar/, then change the environment variable **WPEroot** in your **warprc** file:

```
setenv WPEroot      /usr/foo/bar/
```

2.4.4 The Directory File: wpe.slisp

The directory file **wpe.slisp** binds the environment variables defined in the environment file to the Common Lisp names of files and directories used by WPE The directory file is located in **$WPEhome/current/wpe.slisp** and **$WPEhome/exp/wpe.slisp**, respectively.

2.4.5 The Configuration File: wpeprofile.slisp

When loading a customized environment, the file **wpeprofile.slisp** determines the initial configuration of the Warp Programming Environment. For example, the file specifies whether the W2 compiler is initially loaded (***load-compiler***), whether garbage collection is done noisy or silently (***gc-silence***), whether the core image can be saved (***load-core-image***), etc. The configuration file is looked up in three places until a file is found: First in the current directory, then in the user's home directory and finally in the directory **$WPEhome/**. The file **$WPEhome/wpeprofile.slisp** contains the default configuration. If you want to specify your own configuration, copy it into your home-directory or current directory and modify it.

2.4.6 The Update File: bugfixes.slisp

Sometimes a bug has already been fixed by the maintainer, but has not yet been incorporated into a released version. The purpose of the update file is to deal with those bugs. In general, the update file will contain commands to load some Lisp code that contains the bug fix. It can also be used to temporarily overwrite default settings of environment variables. For example, it could contain a command to set the Warp host to "warpb", if the default Warp host is down because of hardware problems. The update file is looked up in two places, the current directory

or **$WPEhome/current/**[27]. It controls whether the Warp shell should handle all errors that occur during command execution. If 'yes' (the default), the shell handles errors by printing out an error message and simply continuing. If 'no', no handling occurs and the Common Lisp debugger is called instead. This makes it possible to use the normal Common Lisp debugging mechanisms to debug the Warp Programming Environment. Note that this command has no effect when executing Common Lisp expressions inside the Warp shell. In this case the Common Lisp debugger is always called if an error occurs.

2.5.2 LOAD

```
LOAD          load a lisp file
SYNTAX:       load
OPTIONS:
 -silent: Do not warn user about redefinitions  (Default) [Switch]
 -verbose: Warn user about redefinitions [Switch]
EXAMPLES:
 load foo.slisp -verbose
 load foo.lisp
 load foo
```

This command can only be executed when you are in implementor mode (see W2-SET). LOAD permits the user to load a Common Lisp file. It is equivalent to the Lisp 'load' function, except that one does not have to type any parentheses or quotes. The extension is optional. If the extension is missing, the default extensions are looked up in this order:'.*.slisp', '*.lisp'.

When a LOAD command is executed the first time, the version number in the mode line of the Warp shell buffer is extended by a version increment number (initially **.1**) and marked with a "*". For every following LOAD command the version increment number is increased by 1.

2.6 Writing Warp Shell Commands

The functionality provided by the Warp shell might not be sufficient for the needs of the user. In the following I give a rough idea of how to program the shell[28]. **WARNING**:This section is written for users who want to customize their shell. The description applies to the Lisp shell version 2.3.2 of November 2, 1987. Send mail to wpe@sam before you start writing your own shell commands to find out about any changes in the Lisp shell. **END OF WARNING.**

---

[27]or **$WPEhome/exp/ if you are using the experimental version**, respectively, until it is found. If the update file is found in the current directory, say /usr/foo/, the following message is printed at startup:

```
Reading bug fixes from file "/usr/foo/bugfixes.slisp"...
```

The update file is executed after all WPE files have been read, but before the Warp shell initial command file **warpshellinit.cmd** is executed. It should not be used by any other person than the current maintainer of the Warp Programming Environment.

2.5 Special Warp Shell Commands

The following Warp shell commands are useful when you are programming and debugging the Warp shell.

2.5.1 HANDLE-ERRORS

```
HANDLE-ERRORS Handle errors that occur within commands
SYNTAX:       handle-errors [no]
```

This command is automatically executed when you switch to implementor mode (see W2-SET

[28]A more complete description is contained in *Dario Giuse, "Programming the Lisp Shell", Carnegie Mellon University, October 1985.*

The implementation of a Warp shell command consists of two constructs: A definition which defines the functionality of the command, and an entry in the command descriptor table which describes the command.

Let's say we want to write a Warp shell command that controls the echoing of all commands being executed by the Warp shell. We would like to provide an option that restricts the echoing to noninteractive commands. The Common Lisp code to achieve this is shown in figure 2-2 and we will now briefly discuss how that is done.

```
(in-package 'shell)
(defun sh-Noisy (&rest arguments)
  (let ((BatchOnly (member :batchmode arguments))
        (ON (string-equal (First arguments) "ON"))
        (OFF (string-equal (First arguments) "OFF")))

    ;;;argument checking
    (unless (or ON OFF)
      (format t "argument must be ON or OFF~%")
      (return-from sh-Noisy nil))

    (setf Noisy-Switch
          (and ON (not BatchOnly)))
    (setf Batch-Noisy-Switch ON)
  ))

(create-entry
 *commands-table* "Noisy" 'sh-Noisy
 "Controls echoing of shell commands before they are executed." "Noisy [on|off]"
 " 'Noisy on' turns echoing on.    'Noisy off' turns echoing off.   Echoing
  means that a shell command is echoed before it is executed.
  (The expanded command itself is not printed.   This is controlled by
  the verbose command)."
 '(options .   ((:batchmode nil nil
                 "Echo only noninteractive commands ")))
 '(see-also .   (verbose))
 '(examples .   (       "noisy on -batchmode"))
 '(argument-count .   (0 1)))
```

Figure 2-2: Warp Shell: Implementation of the Command NOISY

The shell command must be defined in package **'shell**. The definition of the command consists of two parts: a function definition, **sh-Noisy**, defining the functionality of the **Noisy** command, and a call to the function **create-entry** which sets up a command descriptor for the Noisy command. **sh-Noisy** is passed a parameter **arguments** from the Warp shell. **arguments** is determined from the user input as follows: The first complete word of the user input is interpreted as the name of the command to be invoked. This first word is not passed in arguments. Words that start with a minus sign are considered as options and are converted to keywords, i.e. symbols interned in the Common Lisp KEYWORD package. The name of the symbol corresponds to the whole option name, with the minus sign replaced by a colon sign. All remaining words are converted to strings, without further interpretation. The result of this parsing is a list of elements, either strings or keywords, passed to the Lisp function. For example, if the user types '**noisy on -batchmode**', the Warp shell issues the call '**(sh-noisy "on" :batchmode)**'.

The **sh-Noisy** function is implemented as follows. First it checks **arguments** for the occurrence of the keyword **:batchmode** and stores the result in the local variable **BatchOnly**. Then it does some type checking on the argument list (this is only necessary because the Lisp shell does not do any type checking yet). The remainder of **sh-noisy** is very simple. Echoing of commands is controlled by the two global variables **Noisy-Switch** and **Batch-Noisy-Switch** which are set accordingly to whether "ON" or "OFF" was passed in **arguments**.

The command descriptor provides the link between the Lisp function and what the user types. It also provides information for the help facility type checking of arguments (the latter is not yet implemented). A command

descriptor entry consists of several parts:

- *commands-table* is the name of the command descriptor table.
- User string. This is the name of the command as typed by the user.
- Lisp Function Name. The name of the lisp function to be called if the command line input starts with the user string.
- One-line command description. This is a short summary of the main function performed by the command. This one-line help is printed by the generic help function as the only description of a command; it is also printed by the extensive command-specific help as the header. This information is also the only piece of information about a command (besides the command name) examined by the KEYWORD command.
- Syntax. This is a one-line entry that describes the syntax for a given command. This entry follows the same rules that are used in the UNIX manual; for instance, optional parameters are listed in square brackets. This information is printed by the specific help for a command in the SYNTAX section.
- Help. This is an extensive, multi-line description of the command. This entry describes not only the general behavior of the command, but also all the specific details about its unusual aspects or about specific options.
- Options. This list of lists describes the options the command can take. Each option is a list consisting of the full option name, followed by the number of parameters the option can take, followed by the default value of the option (if any) and finally a short explanation of what the option does.
- See-Also. This is a list of other Shell commands that are related to the command being described, and is printed by the extensive command help.
- Argument-Count. This is a list of two values specifying the minimum and maximum number of parameters of the command.
- Examples. This is a set of examples that are meant to illustrate the usage of a command. If the shell command is used within an editor, the example allow for easy exploration of the command language, because they can be directly fed to the shell command interpreter.

For example, after the definition of the noisy command has been loaded into the Warp Shell and the user types **help noisy**, the following output will appear on standard output:

```
NOISY        Controls echoing of shell commands before they are executed.
SYNTAX:      Noisy [on|off]
'Noisy on' turns echoing on.    'Noisy off' turns echoing off.   Echoing
means that  a shell command is echoed before it is executed.
(The expanded command itself is not printed.   This is controlled by
the verbose command).
OPTIONS:
 -batchmode: Echo only noninteractive commands
SEE ALSO:    verbose
EXAMPLES:
 noisy on -batchmode
```

2.7 Debugging the Warp Shell

If HANDLE-ERRORS NO has been issued and the Warp Shell encounters a bug, the Common Lisp debugger is entered. The debugger can also be entered by a keyboard interrupt. If you are using Gosling's Emacs, call the Mock Lisp command SEND-INT-SIGNAL. If you are using GNU Emacs, type CTL C twice.

2.8 Using the Warp Shell: An Example

In this section we walk through a session using WPE and explain the use of some of the Warp shell commands. The session contains commands to set a breakpoint and inspect some variables.

2.8.1 Example Session

In this example we are starting WPE by typing **wpeg** to the UNIX shell. Various comments will appear on the screen during the loading process:

```
> "Gnu-Emacs"
> "18.36.6"
>  Warp Programming Environment 2.6, 08-Jan-88
Startup command file "/usr/wpe/warpshellinit.cmd"...
Type help for help
@% w2-set -host warpm
Warp machine set to "Excalibur"
Warp host set to "warpm"
%
```

The @ sign in front of the prompt sign (**@%**) indicates that a command file - in this case **warpshellinit.cmd** - is executed. The single prompt sign **%** indicates that we can start typing commands. Let us assume we want to compile the W2-program **poly.w2** shown in figure 2-3 and that the program is located in directory **/usr/wpe/hot/test/poly/**.

```
% path /usr/wpe/hot/test/poly/
% w2-compile -file poly.w2 -debug
Compiling: poly.w2
Opt: Local - Off. Global - Off.
All addresses are computed locally on CELL
Parsing and semantic analysis
Compiling for cells 0 through 9
Processing function #0: INIT
Generating cell code
     >>  Execution time on a cell = 1.22E-5 s (61 cycles)
Processing function #1: POLY
Generating cell code
     >>  Execution time on a cell = 4.654E-4 s (2327 cycles)
Generating host and iu codes
  Processing function #0: INIT for last cellprogram
  Processing function #0: INIT for first cellprogram
     >> Execution time on IU = 2.3999999E-6 sec. [12 cycles]
  Processing function #1: POLY for last cellprogram
  Processing function #1: POLY for first cellprogram
     >> Execution time on IU = 4.1199997E-5 sec. [206 cycles]
Total W2 compilation time : 0 minutes 33 seconds
Post-Processing...
Post-processing completed successfully.
Compilation of /usr.WARP/wpe/hot/test/poly/poly.w2 finished.
```

When the w2-compiler has finished the compilation of the program we define two Warp shell types **float10** and **float100**:

```
% w2-type float10 array[10] of float
Type "float10" created
% w2-type float100 array[100] of float
Type "float100" created
% w2-show -types
Warp Shell Types:
     Name            Basetype        Bounds          Size(Bytes) Display
     float10         float           (10)            40
     unsigned-byte   byte            ---             1
     char            char            ---             1
     float           float           ---             4
     unsigned char   char            ---             1
     integer         int             ---             4
     float100        float           (100)           400
     signed-byte     byte            ---             1
     signed char     char            ---             1
     byte            byte            ---             1
     int             int             ---             4
```

```
module polynomial (data in, polycoeffs in, results out)
float data[100], polycoeffs[10];
float results[100];

cellprogram(cid : 0 : 9)

begin

float coeff;

function init

  begin

  int i;
  float temp;

/*Every cell saves the first coefficient that
reaches it, consumes the data and passes the
remaining coefficients.   Every cell generates
an additional item at the end to conserve the
number of receives and sends.                */

  receive(L, X, coeff, polycoeffs[0]);
  for i:= 1 to 9 do
    begin
    receive(L, X, temp, polycoeffs[i]);
    send(R, X, temp);
    end;

  send (R, X, 0.0);
  end

  function poly

    begin

    float xin, yin, ans[100]; /* temporaries */
    int i;
    int j;

/* Implementing Horner's rule, each cell
multiplies the accumulated result yin with
incoming data xin and adds the next
coefficient                                  */

     j := 0;
     for i := 0 to 100 do begin
            j:= j + 1;
            receive (L, X, xin, data[i]);
            receive (L, X, yin, 0.0);
            send (R, X, xin);
            ans[j] := coeff + yin*xin;
            send (R, X, ans[j], results[i]);
        end;
   end

   call init;
   call poly;

end
```

Figure 2-3: W2 Example: Polynomial Evaluation using the Horner Scheme

Furthermore we define 3 Warp shell variables named **data**, **polycoeffs** and **results**:

```
% w2-var -name data -type float100 -init -text /usr/wpe/hot/test/poly/data -comp c2m2
Connecting to Warp host "warpm"...
...initializing...
Variable "data" created in component "c2m2"
% w2-var -nam polycoeffs -ty float10 -tex /usr/wpe/hot/test/poly/polycoeffs -comp c2m2
Variable "polycoeffs" created in component "c2m2"
% w2-var -name results -type float100 -text /usr/wpe/hot/test/poly/results
Variable "results" created
% w2-show -var
Warp Shell Variables:
    Name        Type        Memory   MDesc Mode  Init'd File
    polycoeffs  float10      c2m2     1     text  no     /usr/wpe/hot/test/poly/polycoeffs
    results     float100     d/c      2     text  no     /usr/wpe/hot/test/poly/results
    data        float100     c2m2     0     text  yes    /usr/wpe/hot/test/poly/data
```

We would like to set a breakpoint in function `init`, but we do not exactly know where, so we ask the shell to suggest some:

```
%   w2-suggestbreaks -line 25 30 -function init
Possible  breakpoints in function INIT (source line range 25 to 30):
    Node:13 Line:25 Addr:(276) Type:$DYADIC$ Op:$I-PLUS
    Node:14 Line:25 Addr:(278) Type:$STORE Name:INIT$I
    Node:4 Line:25 Addr:(266) Type:$STORE Name:INIT$I
    Node:7 Line:27 Addr:(272) Type:$STORE Name:INIT$TEMP
    Node:6 Line:27 Addr:(270) Type:$MONADIC$ Name:INIT$TEMP Type:FLOAT Op:$RECEIVE
    Node:9 Line:28 Addr:(272) Type:$MONADIC$ Name:INIT$TEMP Type:FLOAT Op:$SEND
```

After deciding to break on line **28** we type:

```
% w2-break -line 28 -function init
Breakpoint 1 entered into break table:
Name: 1    ENABLED              Source Line:  28    in Function:  INIT
          Condition:    T
          Action:       ( "w2-halt" )
          Calls:        (0 1 2 3 4 5 6 7 8 9)
          Node:         9    Name:INIT$TEMP Type:FLOAT Op:$SEND
          W1-Address:   (272)
```

Now the program can be executed on the Warp machine. We issue a W2-EXECUTE command which by default locks the Warp machine[29]:

```
% w2-execute -parameters data polycoeffs results
Copying input parameters into cluster memory...
Trying to lock Warp Excalibur...
Warp server queue is empty.
Warp machine is yours:
Downloading cluster and micro code for poly.w2..
Starting execution of module...
Breakpoint 1 encountered in cell 0: Line 28 in "poly.w2"
Executing breakpoint actions...
@break1% w2-halt
```

Once Warp is allocated, the microcode for the clusters, the interface unit and for the Warp cells is automatically downloaded. When the breakpoint is encountered in cell 1, the condition is checked. It evaluates to T (=TRUE) and the action part of the breakpoint is executed, which is W2-HALT. Now the user types: W2-GET -LOCALS -CELLS 1 to get the values of the locals of the current function cells 1. The function **INIT** has just two locals **I** and **TEMP**:

---

[29]Note that if the program was not compiled with the -debug option you would get the following error message:

```
? Warning: /usrw61/bob/wpe/test/poly/poly.w2 was not compiled with -debug option.
  Breakpoints cannot be set.
```

and the execution would proceed without the breakpoints.

```
break1% w2-get -locals -cells 1
[Cell 0]:
  Locals in function INIT:
    I =  1
    TEMP =  0.21
[Cell 1]:
  Locals in function INIT:
    I =  1038174126
    TEMP =  0.61
```

At this point we decide to delete the breakpoint and start the execution again:

```
break1% w2-delete -break 1
Breakpoint 1 deleted
break1% w2-unlock
Unlocking Warp Excalibur...
% w2-execute -parameters data polycoeffs results
Copying input parameters into cluster memory...
Downloading microcode from directory "/usrw61/bob/wpe/test/poly/"
Starting execution of module...

?User server: User server died.
Invalidating memory for all defined Warp shell variables.
```

Apparently there is a problem with one of the components of the Warp Programming environment. We use the W2-SANITY command to find out that the Warp host "warpm" is down:

```
% w2-sanity
Checking state of Warp host "warpm", Warp server, WPE server and Warp array...
---> Warp host is not up.
```

At this point we could wait until "warpm" is up again. Instead, we switch to the other Warp host "d2" and try the execution once more:

```
% w2-set -host d2
Warp machine set to "Galileo"
Warp host set to "d2"
% w2-execute -parameters data polycoeffs results
Reallocating memory for "data" in cluster "c2m2"...
Reallocating memory for "polycoeffs" in cluster "c2m2"...
Reallocating memory for "results" in cluster "dk"...
Copying input parameters into cluster memory...
Trying to lock Warp Galileo...
Warp server queue is empty.
Warp machine is yours:
Starting execution of module...
Execution completed.
Unlocking Warp GE...
```

We now look at the value of the Warp shell variable results:

```
% w2-show -var results -range 0:3
```

which will display the first four elements of the variable result in an editor buffer called **results**:

```
1023.000000 1023.000000 1023.000000 1023.000000
```

2.8.2 Example Command File

The following is a summary of all the commands we have typed in the previous example session:

```
path /usr/wpe/hot/test/poly/
w2-compile -file poly.w2 -debug
w2-type float10 array[10] of float
w2-type float100 array[100] of float
w2-show -types
w2-var -name data -type float100 -init -text /usr/wpe/hot/test/poly/data -comp c2m2
w2-var -nam polycoeffs -ty float10 -tex /usr/wpe/hot/test/poly/polycoeffs -comp c2m2
w2-var -name results -type float100 -text /usr/wpe/hot/test/poly/results
w2-show -var
w2-suggestbreaks -line 25 30 -function init
w2-break -line 28 -function init
w2-execute -parameters data polycoeffs results
w2-get -locals -cells 1
w2-delete -breaks 1
w2-unlock
w2-execute -parameters data polycoeffs results
w2-sanity
w2-set -host d2
w2-execute -parameters data polycoeffs results
w2-show -var results -range 0:3
```

3. The Warp Monitor

3.1 Introduction

The Warp monitor is intended as a building block for the implementation of tools such as the Warp shell or the W2 debugger as well as for standalone applications using the Warp machine and its individual components.

The main goal of the Warp monitor is to shield the programmer from the complexity of the Warp array and yet make the hardware accessible. The programmer can access the Warp array via a set of diagnostic routines collectively called the *Warp library* and a set of routines called the *Master Host Library*. The Warp Library accesses the interface unit and the cells of the Warp machine via serial chains. The Master Host Library provides control over the cluster processors[30]. Both libraries assume that the Warp machine is used by only one user.

The abstractions provided by the Warp monitor are more complete and on a higher level than these libraries. The Warp monitor provides access to the Warp array as well as to the cluster processors. The goal is to provide the programmer with a "virtual Warp machine": The Warp monitor provides a set of functions to acquire and release the Warp machine, to allocate memory in the Warp host and cluster memories, to download Warp micro code and cluster code, to control the execution of Warp programs and to monitor their behavior.

Figure 3-1 shows the architecture of applications using the Warp monitor. The Warp machine level consists of the **Cluster Processors**, **Interface Unit** and the **Warp Array** (see also Figure 1-1). Above the hardware level is the first software level implemented by the **Warp Library**, the **Master Host Library** and the **Warp Monitor**. The second software level implements the application programs. In the figure, Application 1 is a standalone program calling the Warp monitor functions directly. Application 2 is an application program written in C built on top of the Warp User Package. Application 3 is an application written in Common Lisp running on top of the Warp shell. And as we can see, the **Warp shell** alone is also an example of an application program.

3.2 Classification of Warp Monitor Functions

The Warp monitor functions are grouped in several parts: Warp server control, error handling, Warp locking mechanism, event flags, memory allocation, downloading, execution, debugging and miscellaneous functions.

The *Warp server control functions* provide control over the Warp server and the user servers. `start_warpd` starts or restarts the Warp server. The `openconn` function supports multiple Warp machines. It permits the user to select a particular Warp host on which to run the application and the type of user server to be used for the application. There are four different types of user servers. Server type 0 provides only the Warp locking mechanism functions (`lock_warp`, `unlock_warp`, `list_queue` and `next_entry`) needed to support multiple users. Server type 1 provides all the Warp monitor functions documented in this section via TCP/IP to a user server process. Server type 2 is an experimental server used for development purposes. Server type 3 is similar to Server type 1, except that it knows whether the application is running in local or remote mode. If in remote mode, a user server process is started up on the Warp host side and the communication between application and user server will be via TCP/IP. If in local mode, only the Warp server is a separate process providing the Warp locking mechanism functions. All the other Warp monitor functions are linked into the application program and are executed directly. Note that in local mode the cluster memory, that is about 8 Mbytes, is mapped into the memory space of the application program. This causes problems for the Lisp interpreter, because it should not be managed

[30]See *Warp Library User Manual* and *User Manual for the Warp Host Software*, both by General Electric, Radar Systems Department, Syracuse, for more details.

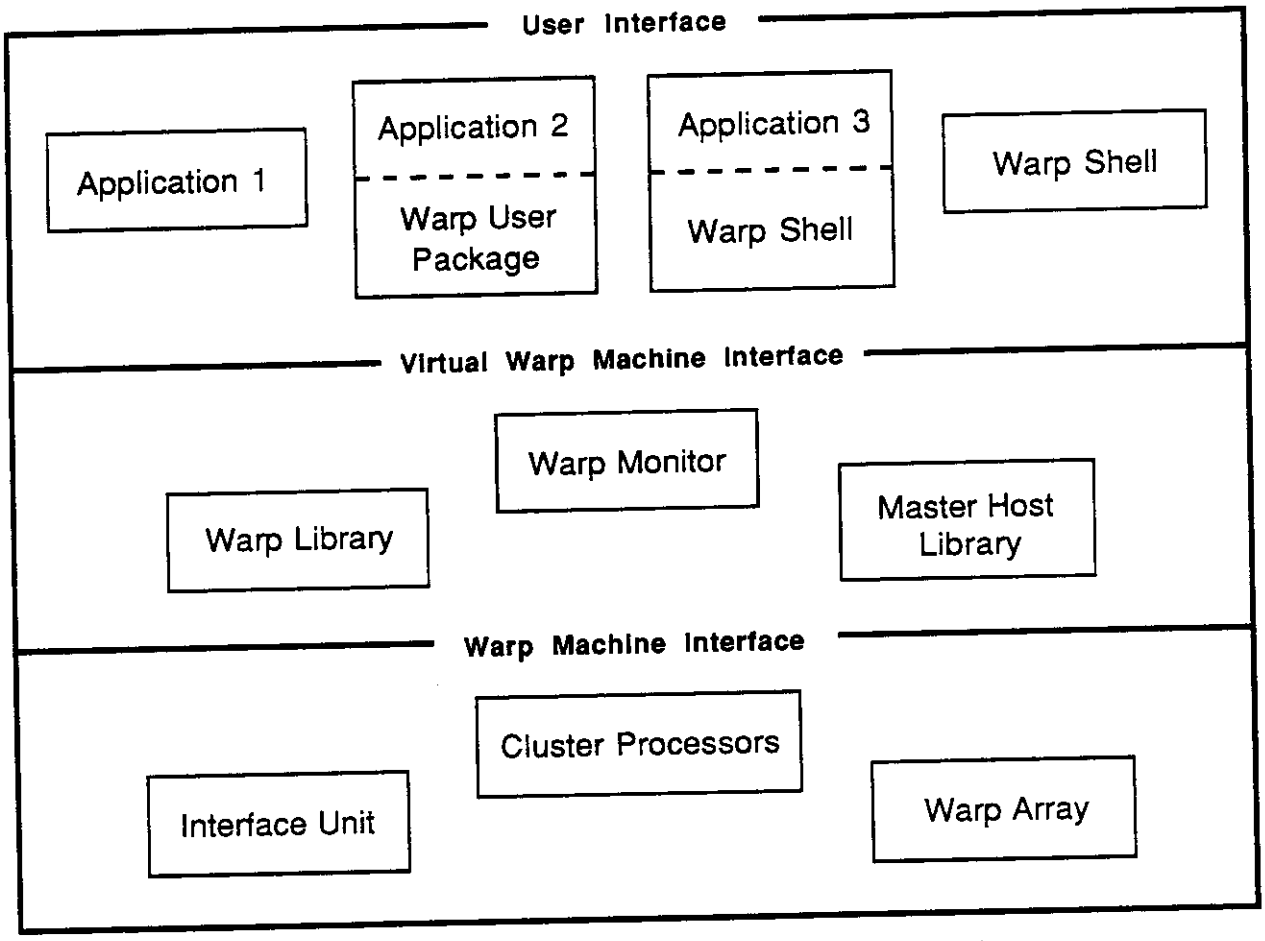

Figure 3-1: Architecture of Applications using the Warp Monitor

by Lisp's memory manager. Thus, for Lisp applications, server type 1 should be used, whereas for real-time applications server type 3 should be the preferred server.

Error handling is done by the functions `server_error` and `get_error_string`. Most of the Warp monitor functions return a 0 if the call was successful and a 1 if it was not successful. The function `server_error` can be called which returns an index into a string table that stores all possible error messages. `get_error_string` returns the corresponding string.

The *Warp Locking mechanism* is provided by the functions `lock_warp`, `unlock_warp`, `list_queue` and `next_entry`. These functions control the queue of users who have requested the use of the Warp machine. The `lock_warp` checks whether the queue is empty and if yes, it enters the user into the queue and locks the machine for him. If it is not empty, the user is appended at the tail of the queue. The `unlock_warp` checks whether the queue is empty and if yes, it takes the user from the head of the queue, therefore making the machine accessible for the next queue entry. The functions `list_queue` and `next_entry` permit the inspection of the queue.

One of the purposes of the Warp monitor is to permit programmers to view the Warp machine as a sequential machine. However, it is also possible to view the Warp machine as a set of five processors. The programmer can execute concurrent tasks on the Warp array (`start_warp`), the two cluster processors and the support processor (`start_cluster`) and on the Warp host. *Event flag functions* control the execution of these tasks. `alloc_ef`

and `free_ef` manage the allocation of event flags. The `set_ef` function sets an event flag to one of the following states: PENDING, INPROGRESS, DONE, ERROR and UNUSED. Event flags states are also set by the processors. For example, when a processor starts a service associated with a specific event flag, it changes the status to INPROGRESS. When the service has been completed, the status is changed to DONE. The `wait_ef` function permits the programmer to wait for an event flag to reach a certain state. An event flag is basically a small integer which serves as an index into an event table. The event table is stored in the memory of the support processor. Whenever the Warp machine is unlocked, the event table is swapped into the user server address space, and when the Warp machine is locked again, the event table is swapped back into the support processor. Thus event flags are valid across several locks/unlocks of the Warp machine.

The *memory allocation functions* provide the user with the ability to allocate and deallocate memory on the cluster processors (`get_uninit_llmem`, `alloc_clmem`, `dealloc_clmem`, `free_buffer`) and copy data between the application program's address space and the cluster memory (`read_from_clmem`, `write_to_clmem`) or between files and cluster memory (`copy_ctc`, `copy_ftc`, `copy_ctf`). User server and cluster processor memory is represented by so called *memory descriptors* which are basically small positive integers. File names are always interpreted in the naming context of the file system used by the Warp monitor, not by the application.

Downloading functions load micro code into micro code memory (`load_micro`) or load cluster code into cluster processor memory (`load_cluster`, `load_one_cluster`). Downloading functions using event flags are of advantage if a real-time application program makes use of more than one Warp program. Such applications can cache the cluster code (`make_clust_func`) and microcode (`cache_micro`) for all Warp programs in Warp host memory. When switching from one Warp program to another, microcode can be quickly downloaded into the Warp array with `fast_load_micro`. Note that `load_micro` or `fast_load_micro` have to be called every time before a program is executed on Warp, even if the the Warp is kept locked during repeated executions of the same program. The reason is that these routines also reset the Warp array.

Execution functions control the execution of the components of the Warp machine. The programmer has the choice to execute the components as a unit or individually. `execute_warp` starts the execution of all the components, that is, the cluster processors as well as the Warp array. `start_cluster` starts a function previously defined with `make_clust_func` on an individual cluster processor. `start_warp` starts the execution of a previously downloaded Warp program on the Warp array but does not touch the cluster processors. `continue_warp` resumes the execution of a Warp program after one or more cells have encountered a breakpoint. `execution_time` returns the time elapsed during the execution of the last W2 program.

Debugging functions permit the user to set breakpoints and to inspect various resources in the hardware. The function `use_printf` permits the user to enable print statements inserted in cluster code programs. All the other debugging functions are concerned with debugging the Warp array. `set_break` and `clear_break` set and delete breakpoints in the micro code in one or more cells. `set_pc` and `get_pc` manipulate the program counters of the interface unit and the cells. There are functions to inspect and write the micro code (`read_data_mem`, `write_data_mem`), to read or write single registers (`read_register`, `write_register`), to read or write serial chains (`read_chain`, `write_chain`) or parts of it (`get_field`, `put_field`).

There are several *miscellaneous functions*: `set_debug` turns on a tracing mechanism to report the message traffic between application and the Warp monitor. `set_timeout` sets the length of time-out for Warp programs. `get_version` returns the version number of the Warp monitor. `set_dir` changes the current directory on the Warp monitor side. The `sanity_check` checks the state of several components of the environment such as the file system, the Warp host, the Warp and WPE servers and the Warp array. If there is any problem with the Warp

array, it can be reset to a defined state with the `reset_warp` command.

The Warp monitor is written in C and therefore the following specifications are also in C. However, application programs using the Warp monitor functions can be written in any language as long as its implementation supports the call of C functions. For example, the Warp shell is written in Lucid Common Lisp, which contains a Common Lisp extension to call C functions.

3.3 Warp Server Control Functions

3.3.1 START_WARPD

Start the Warp server.

Interface:

```
start_warpd()
```

Parameters: None.

Returns: None.

Notes: Normally the Warp server can be assumed to be up and running. This function should be used only if the Warp server is not running. `start_warpd` only attempts to restart the Warp server. There is no guarantee (or indication) of success.

3.3.2 OPENCONN

Open a connection to a Warp host, that is, create a user server process or reconnect to it.

Interface:

```
openconn(host,mode)
char *host;
int mode;
```

Parameters:

`host` The name of the Warp host to (re)connect to, or 0 if a default host should be used. The default host is determined as follows: First `openconn` checks whether a UNIX environment variable `WARPhost` has been defined[31]. If yes, the default host is the first string extracted from `WARPhost` (`WARPhost` can be a list of strings separated by blanks). If not, a *hardwired* name is used[32].

`mode` The type of server subprocess to use. Legal types are:

- 0 = No user server process (use Warp server for queueing only).
- 1 = Normal user server process (use server for all Warp monitor operations).
- 2 = Experimental user server process (for development purposes).
- 3 = If in remote mode, start up a user server process on the Warp host side. In this case the communication between

[31] `WARPhost` is automatically defined if the environment file `$WPEhome/warprc` has been "sourced".

[32] At CMU, the possible Warp host names are "warpb", "warp8" and "warpm". The hardwired default is "warpm" unless the application is running on workstation **warpb**, in which case "warpb" is the default.

application and user server will be via TCP/IP. If in local mode, only the Warp server is a separate process and provides the Warp monitor functions `lock_warp`, `unlock_warp`, `list_queue` and `next_entry`. All the other Warp monitor functions are linked into the application program and are called as direct procedure calls. In this case the cluster memory, that is about 8 Mbytes, is mapped into the memory space of the application program.

Returns: Returns 0 if successful, 1 otherwise. If this call fails, the Warp Host does not exist or it was not up.

Notes: The name of the Warp host must be in lower case. If successful, `openconn` connects to a Warp host, that is, it creates a user server process, or it reconnects to it, if there is already an open connection to that host. `openconn` automatically performs a `set_dir` call to the current directory of the application program. A call to `openconn` does not close any previous Warp connections. Therefore, if you wish to be greedy, you can have several open Warp connections at once. If multiple connections are used, the Warp host mentioned in the last `openconn` call determines the current Warp host. The connection to the current Warp host can be closed with an `unlock_warp(1)` call.

3.4 Error Handling Functions

This section describes the functions that inspect the error state after a previous Warp monitor function call. `server_error` returns an index pointing to a position in an array of strings and `get_error_string` returns that string. Both functions always refer to the last Warp monitor call that was not an error handling function. The error strings are defined by the following C data structure:

```
#ifdef GIL
char *WP_serv_err[] = {
#else
char *serv_err[] = {
#endif
        "No Error",
        "Unknown Error",
        "Garbled Command",
        "Too Few Parameters",
        "Queue Full",
        "Can't Create User Server",
        "User Server Died",
        "User Interrupt",
        "User Interrupt, Closed Connection",
        "Can't Access Directory",
        "Illegal Cluster Memory",
        "Can't Allocate Memory",
        "No Such Memory Descriptor",
        "Memory Descriptor Too Small",
        "Can't Open File",
        "Not Enough Data",
        "WARP Not Locked",
        "No PC Set",
        "Timeout During Event Flag Wait",
        "No Appropriate Cluster Code",
        "Breakpoint Encountered",
        "No Procedure Running",
        "Low Bound Is Larger Than High Bound",
        "No Such Chain",
        "No Such Register",
        "No Such Field",
        "No Such Operation",
        "Field Is Larger Than 32 Bits",
        "No Event Flags Left",
        "Bad Event Flag",
        "Illegal Cluster Function",
        "Error In Reading .param File",
        "Illegal Cluster Memory Descriptor",
};
```

3.4.1 SERVER_ERROR

Return error index of last Warp monitor call.

Interface:

```
int server_error()
```

Parameters: None.

Returns: the error index from the last Warp monitor function call. The error code is an index into the data structure `serv_err` defined above.

Notes: Successive calls to `server_error` don't change the error index.

3.4.2 GET_ERROR_STRING

Get the error string from the last Warp monitor call.

Interface:

```
int get_error_string(buffer)
char *buffer;
```

Parameters:

`buffer` A buffer in which to store a description of the last error.

Returns:

0 if successful, 1 otherwise.

Notes: Successive calls to `get_error_string` don't change the error index.

3.5 Warp Locking Mechanism

3.5.1 LOCK_WARP

Lock the Warp machine so it cannot be accessed by other users.

Interface:

```
int lock_warp(delay, send_ok, comment)
int delay, send_ok;
char *comment;
```

Parameters:

`delay`	0 = don't wait for the lock, 1 = wait in the Warp server queue if somebody else is currently using the Warp machine.
`send_ok`	0 = don't send messages to the terminal, 1 = OK to notify.
`comment`	An identifying string for the Warp server queue.

Returns:

0 if lock was granted and 1 otherwise.

Notes: Users are treated in FIFO order by the Warp server. A message is sent to the current owner of the Warp machine, that is, the head of the Warp server queue, if `delay` is 1. `lock_warp` can also be used to double check if the Warp is locked.

3.5.2 UNLOCK_WARP

Unlock the Warp machine so it can be accessed by others.

Interface:

```
int unlock_warp(finish)
int finish;
```

Parameters:

`finish`	1 = close user server connection, 0 = leave user server connection open.

Returns:

0 if connection is in expected state, 1 if connection had to be closed anyway.

Notes: When the Warp machine is unlocked, memory allocated with `alloc_clmem` is saved into the user servery memory. After the Warp is unlocked, the following is true:

- The event table is swapped from support processor memory to the user server memory.
- Cluster memory described by memory descriptors is swapped from cluster memory to the user server memory.
- Cluster code loaded by `load_cluster` and `load_one_cluster` is invalidated[33].
- Micro code loaded into the Warp array with `load_micro` and `fast_load_micro` is invalidated.
- Cluster code function ID's are invalidated.

[33] *Invalidated* means the user must load the cluster code everytime the Warp machine is locked.

- ID's of micro code loaded with `cache_micro` are still valid.

The `unlock_warp` always succeeds in unlocking the Warp, regardless of the return code. The Warp is automatically unlocked when the process that called `lock_warp` dies.

3.5.3 LIST_QUEUE

Determine the length of the Warp server queue, that is, the number of users that have issued a `lock_warp` call, but not yet an `unlock_warp` call.

Interface:

```
int list_queue()
```

Parameters: None.

Returns: the number of jobs in the queue, or -1 if this cannot be determined.

Notes: `next_entry` must be called after `list_queue` was called.

3.5.4 NEXT_ENTRY

Get next entry from Warp server queue.

Interface:

```
int next_entry(buf)
char *buf;
```

Parameters:

`buf` An array of characters in which the next queue entry will be stored. Each entry has the form: user machine tty "comment"

Returns: 0 if an entry was retrieved, 1 if not.

Notes: `list_queue` must be called before this routine can be called. After `list_queue` is called, you must call `next_entry` repeatedly until it returns 1 (no more queue entries).

3.6 Event Flag Functions

All event flag functions use a type `EFLAG`, which is defined in **$WARPinclude/monitor.h or gilmon.h.**

3.6.1 ALLOC_EF

Allocate event flag.

Interface:

```
EFLAG alloc_ef()
```

Parameters: None.

Returns: An event flag or 0 if no flags are available.

Notes: Obsolete event flags can be deallocated with `free_ef`. For efficiency reasons, the event flag table is stored in the memory of the support processor. When unlocking the Warp machine, the event flag table is copied into Warp host memory. Thus, event flags can also be used when the Warp is not locked. When the Warp locked again, the event flag table is copied back from Warp host memory to support processor memory.

3.6.2 FREE_EF

Free event flag.

Interface:

```
int free_ef(flag)
EFLAG flag;
```

Parameters:

	`flag`	An event flag to be released.

Returns:

An event flag or 0 if no flags are available.

3.6.3 READ_EF

Read event flag.

Interface:

```
int read_ef(flag)
EFLAG flag;
```

Parameters:

	`flag`	An event flag to be read.

Returns:

Status of the event flag (either UNUSED (=0), PENDING (=1), INPROGRESS (=2), DONE (=3), ASYNCDONE (=4) or ERROR (=128)) or -1 if there is an error.

Notes: ERROR is a status from a successful read.

3.6.4 SET_EF

Set event flag to a certain value.

Interface:

```
int set_ef(flag,event)
EFLAG flag;
int event;
```

Parameters:

	`flag`	An event flag to be set.
	`event`	The value of the flag (either UNUSED (=0), PENDING (=1), INPROGRESS (=2), DONE (=3), ASYNCDONE (=4) or ERROR (=128)).

Returns:

0 if successful, 1 otherwise.

3.6.5 WAIT_EF

Wait for event flag to be set to a certain value.

Interface:

```
int wait_ef(flag,event,timeout)
EFLAG flag;
int event, timeout;
```

Parameters:

	`flag`	An event flag to wait for.
	`event`	The value of the flag to wait for (either UNUSED (=0), PENDING (=1),

INPROGRESS (=2), DONE (=3), ASYNCDONE (=4) or ERROR (=128)).

`timeout` The timeout delay (0 = wait forever).

Returns:

0 if successful, 1 otherwise.

Notes: If the `event` value is DONE, this call returns immediately. The `timeout` value is in no recognizable units.

3.7 Memory Allocation

From the point of view of the programmer, there are three kinds of memory: application memory, user server memory and cluster memory. If the application is in local mode, then application and user memory are identical. Cluster memory is only accessible if the Warp machine is locked. Furthermore, cluster memory is reallocated every time the Warp machine is locked. Thus pointers to cluster memory obtained by calling `get_uninit_clmem` are only valid for the time the machine is locked. In order to maintain consistency between remote and local operations, the following rules must be followed when dealing with buffer pointers in application or user server memory that point to cluster memory:

- If the buffer is modified, the cluster memory is not automatically updated. Therefore, `write_to_clmem` must be called to make the same change in the cluster memory.
- Before unlocking the Warp, the cluster memory pointed to by the buffer pointers should be saved into user memory and then the buffer should be freed. Once the Warp machine is unlocked, the buffer pointer are no longer valid (If Warp locked again, those buffer pointer would point to illegal memory because of the reallocation of the cluster memory).

3.7.1 ALLOC_CLMEM

Allocate memory in cluster memory.

Interface:

```
int alloc_clmem(type, size)
char *type;
int size;
```

Parameters:

`type` Where the memory should be allocated (one of "sun", "c1m1", "c1m2", "c1m3", "c2m1", "c2m2", "c2m3", or "dk").

`size` The number of bytes to be allocated.

Returns:

either a "memory descriptor" (a positive integer), or -1 if there is a problem.

Notes: *Memory cannot be allocated in "c1m1" or "c2m1".* When the Warp machine is already locked, memory is allocated in the cluster memory of the Warp machine. If it is not yet locked, it is allocated in the user server memory and downloaded automatically when the user gets the Warp machine. "dk" stands for don't care. Certain errors such as "Server Process Died" invalidate all memory descriptors.

Each cluster memory board contains 1 MByte of memory. Starting with Warp monitor version 4.13, it is possible to allocate cluster memory of up to 2 MBytes in C1 or C2. The cluster memory components c1m2 + c1m3 and c2m2 + c2m3, respectively, have been merged into 2 Mbytes of consecutive memory. The old naming scheme has been kept for compatibility purposes. However, when calling `alloc_clmem` with c1m2 or c1m3 (c2m2 or c2m3), the Warp monitor will always start allocating memory in the c1m2 (c2m2) memory board and allocate memory until it runs out of memory in board c1m3 (c2m3).

3.7.2 DEALLOC_CLMEM

Deallocate memory in cluster memory.

Interface:

```
int dealloc_clmem(desc)
int desc;
```

Parameters:

desc	A memory descriptor returned by ALLOC_CLMEM.

Returns:

0 if memory is freed, 1 if there was some problem.

3.7.3 GET_UNINIT_CLMEM

Get uninitialized cluster memory.

Interface:

```
char *get_uninit_clmem(desc,offset,size)
int desc, offset, size;
```

Parameters:

desc	The memory descriptor to get a pointer to.
offset	Where in the memory descriptor to point to.
size	How many bytes should be pointed to.

Returns:

0 if there was a problem with the descriptor and a pointer to an uninitialized buffer for the cluster memory otherwise.

Notes:

When in local mode, then `get_uninit_clmem` returns a pointer to the cluster memory described by memory descriptor `desc` and `offset`. When in remote mode, then `get_uninit_clmem` calls `malloc` to create a buffer in the client address space that will be used to hold the cluster memory described by memory descriptor `desc` and `offset` in the server address space and returns a pointer to that buffer. As a result, don't expect to read anything useful out of this buffer when in remote mode. When the Warp machine is already locked, memory is allocated in the cluster memory of the Warp machine. If it is not yet locked, it is allocated in the user server memory and downloaded automatically when the user gets the Warp machine.

3.7.4 READ_FROM_CLMEM

Copy from cluster memory into a buffer.

Interface:

```
char *read_from_clmem(desc,offset,size,buf)
int desc, offset, size;
char *buf;
```

Parameters:

desc	The memory descriptor to read from.
offset	Where in the memory descriptor to start reading from.
size	How many bytes to read.
buf	A buffer to hold the data or 0 if a buffer should be allocated.

Returns:

0 if there was a problem with the read and a buffer pointer otherwise.

Notes:

When in local mode and `buf = 0`, `read_from_clmem` returns a pointer to the cluster

memory described by memory descriptor `desc` and `offset`. When in remote mode and `buf` = 0, `read_from_clmem` calls `malloc` to create a buffer and reads over the network the cluster memory described by memory descriptor `desc` and `offset` into that buffer. When in either remote or local mode and `buf` <> 0, `read_from_clmem` copies the cluster memory described by `desc` and `offset` into the buffer described by `buf`.

3.7.5 WRITE_TO_CLMEM

Copy from a buffer to cluster memory.

Interface:

```
int write_to_clmem(desc,offset,size,buf)
int desc, offset, size;
char *buf;
```

Parameters:

`desc`	The memory descriptor to write to.
`offset`	Where in the memory descriptor to start writing.
`size`	How many bytes to write.
`buf`	A buffer holding the data to be written.

Returns:

0 if the write was successful, 1 if there was some problem.

Notes:

When in local mode and `buf` is pointing to cluster memory, then `write_to_clmem` is a NOOP. When in local mode and `buf` does not point into cluster memory, then the memory contained pointed to by `buf` is copied to cluster memory described by `desc` and `offset`. When in remote mode, then `write_to_clmem` copies the buffer over the network from the buffer described by `buf` to the cluster memory described by `desc` and `offset`.

3.7.6 FREE_BUFFER

Free memory allocated by `read_from_clmem` or `get_uninit_clmem`.

Interface:

```
free_buffer(buf)
char *buf;
```

Parameters:

`buf`	A buffer allocated by `read_from_clmem` or `get_uninit_clmem`.

Notes:

When in local mode, a call to `free_buffer` is a NOOP, because there is no buffer to free. When in remote mode, the memory described by `buf` is freed (by calling `free`).

3.7.7 COPY_CTF

Copy cluster memory into a file.

Interface:

```
int copy_ctf(desc,offset,size,filename,type)
int desc, offset, size;
char *filename, *type;
```

Parameters:

`desc`	The memory descriptor to copy from.
`offset`	Where in the memory descriptor to start copy from.

`size`	How many bytes to copy.	
`filename`	The name of the file to store the data in.	
`type`	The format to use, one of:	
	"Byte"	8 bit decimal integers.
	"Char"	Binary.
	"Hex"	32 bit hexidecimal integers.
	"Int"	32 bit decimal integers.
	"Float"	32 bit floating point numbers.

Returns:

0 if successful, 1 otherwise.

3.7.8 COPY_FTC

Copy a file into cluster memory.

Interface:

```
int copy_ftc(desc,offset,size,filename,type)
int desc, offset, size;
char *filename, *type;
```

Parameters:

`desc`	The memory descriptor to copy to.	
`offset`	Where in the memory descriptor to start writing.	
`size`	How many bytes to copy.	
`filename`	The name of the file containing the data.	
`type`	The format to use, one of:	
	"Byte"	8 bit decimal integers.
	"Char"	Binary.
	"Hex"	32 bit hexidecimal integers.
	"Int"	32 bit decimal integers.
	"Float"	32 bit floating point numbers.

Returns:

0 if successful, 1 otherwise.

3.7.9 COPY_CTC

Copy cluster memory to cluster memory after event flag is done.

Interface:

```
EFLAG copy_ctc(from,to,size,proc,efnc)
int from, to, size, proc;
EFLAG efnc;
```

Parameters:

`from`	The source memory descriptor.
`to`	The destination memory descriptor.
`size`	How many bytes to copy.
`proc`	Who should do the copy (0 = MASTER1, 1 = CLUSTER1, 2 = CLUSTER2, and 3 = SUPPORT1).

`efnc` An event flag to be set to DONE before the copy is done. In case there is nothing to wait for, `efnc` can be replaced with the constant NOWAIT[34].

Returns:

An event flag if the function is started, 0 otherwise.

3.8 Downloading Functions

3.8.1 LOAD_MICRO

Load Warp micro code into the interface unit and the Warp array.

Interface:

```
int load_micro(file)
char *file;
```

Parameters:

`file` The simple filename of the microcode (no pathname prefix, no extension) to be loaded.

Returns:

0 if successful, 1 otherwise.

Notes:

load_micro has to be called every before executing a program on Warp. The filename is looked up in the directory set by SET_DIR. Loading new microcode invalidates the result of the last `set_pc` call. This call also attempts to read the ".param" file to find the starting addresses of the warp cells and the interface unit. If this is successful, the program counters are initialized to these values. Note that `load_micro` or `fast_load_micro` have to be called every time before a program is executed on Warp, even if the if the Warp is kept locked during repeated executions of the same program. The reason is that these routines also reset the Warp array. However, the routines are clever enough to check whether the micro code has already been downloaded.

3.8.2 CACHE_MICRO

Load W1 micro code file into Warp host memory.

Interface:

```
int cache_micro(file)
char *file;
```

Parameters:

`file` The filename of the microcode (without extension) to be cached into the Warp host memory.

Returns:

A "microcode ID" (a positive integer) if successful, -1 otherwise.

Notes:

This call doesn't touch the Warp machine.

[34]The constants MASTER1, CLUSTER1,CLUSTER2, SUPPORT1 and NOWAIT are defined in the include file **$WARPinclude/Wconfig.h** (See page 69)

3.8.3 FAST_LOAD_MICRO

Download cached W1 micro code from Warp host into Warp array after event flag is done.

Interface:

```
EFLAG fast_load_micro(id,efnc)
int id;
EFLAG efnc;
```

Parameters:

`id`	A "microcode ID" returned from `cache_micro` identifying microcode to be loaded into the Warp array.
`efnc`	An event flag to be set to DONE before the micro code is loaded. In case there is nothing to wait for, `efnc` can be replaced with the constant NOWAIT.

Returns: An event flag if the load is started, 0 otherwise.

Notes: `load_micro` or `fast_load_micro` have to be called every time before a program is executed on Warp, even if the if the Warp is kept locked during repeated executions of the same program. The reason is that these routines also reset the Warp array. However, the routines are clever enough to check whether the micro code has already been downloaded.

3.8.4 LOAD_CLUSTER

Load the files containing cluster code for the two cluster processors.

Interface:

```
int load_cluster(file)
char *file;
```

Parameters:

`file`	The filename of the cluster code (no pathname prefix, no extension) to be loaded.

Returns:

0 if successful, 1 otherwise.

Notes: The filename is looked up in the directory set by `set_dir`.

3.8.5 LOAD_ONE_CLUSTER

Load one file containing cluster code.

Interface:

```
int load_one_cluster(file,proc)
char *file;
int proc;
```

Parameters:

`file`	The filename of the cluster code (with extension!) to be loaded.
`proc`	The processor to load it into: • 0 = Cluster Processor 1 • 1 = Cluster Processor 2 • 2 = Support Processor

Returns:

0 if successful, 1 otherwise.

3.8.6 MAKE_CLUST_FUNC

Load cluster code, create function and return "function code ID".

Interface:

```
int make_clust_func(file)
```

Parameters:

`file`	The filename of the cluster code (with extension!) to be loaded.

Returns:

This call returns a "function code ID" (a positive integer) if successful, or -1 otherwise.

3.9 Execution Functions

3.9.1 EXECUTE_WARP

Execute a Warp program by starting the cluster processors and the Warp machine.

Interface:

```
int execute_warp(cycles,nin,nout,parin,parout)
int cycles, nin, nout;
int parin[], parout[];
```

Parameters:

`cycles`	How many cycles to run (-1 = run to completion or until breakpoint is encountered).
`nin`	The number of input parameters. The maximum of input parameters is 30.
`nout`	The number of output parameters. The maximum of input parameters is 30.
`parin`	The input parameters (memory descriptors from `alloc_clmem`).
`parout`	The output parameters (memory descriptors from `alloc_clmem`).

Returns:

0 if the execution completes, 1 otherwise.

Notes:

`execute_warp` starts the execution of a Warp program consisting of cluster code for the cluster processors and micro code for the Warp machine. If `cycles` is not equal to -1, breakpoints will be ignored. It can only be called after the micro code and the cluster code have been loaded and memory has been allocated for the input and output parameters. Each parameter is assumed to be a memory descriptor returned by `alloc_clmem`. However, 30 bit integers may also be passed by encoding them using the `param_const` function. (Two bits are used as a tag.) These constants will be sign-extended to 32 bits when actually used. `execute_warp` expects the program counter for the warp cells and the interface unit to be set, either automatically by calling `load_micro` or explicitly by calling `set_pc`.

3.9.2 START_CLUSTER

Start cluster processor given a *list* of parameters after event flag is done.

Interface:

```
EFLAG start_cluster(proc,func,efnc,p1,p2,...,-1)
int proc, func, p1, p2, ...;
EFLAG efnc;
```

Parameters:

`proc`	The processor to start running: 1 = CLUSTER1, 2 = CLUSTER2, and 3 = SUPPORT1.

`func`	The "function code ID" of the function to execute (may either be a built-in function or the result of a `make_clust_func` call).
`efnc`	An event flag to be set to DONE before the processor is started. In case there is nothing to wait for, efnc can be replaced with the constant NOWAIT.
`p1, p2,...`	A list of parameters to the function.
`-1`	Indicator to terminate the list of parameters.

Returns:

An event flag if the function is started, 0 otherwise.

Notes:

Each parameter is assumed to be a memory descriptor returned by `alloc_clmem`. However, 30 bit integers may also be passed by encoding them using the `param_const` function. (Two bits are used as a tag.) These constants will be sign-extended to 32 bits when actually used.

3.9.3 START_CLUSTER_A

Start cluster processor given an *array* of parameters after event flag is done.

Interface:

```
EFLAG start_cluster_a(proc, func, efnc, params)
int proc, func, params[];
EFLAG efnc;
```

Parameters:

`proc`	The processor to start running: 1 = CLUSTER1, 2 = CLUSTER2, and 3 = SUPPORT1.
`func`	The function to execute (may either be a built-in function or the result of a `make_clust_func` call).
`efnc`	An event flag to be set to DONE before the processor is started. In case there is nothing to wait for, `efnc` can be replaced with the constant NOWAIT.
`params`	A list of parameters to the function, terminated by -1.

Returns:

An event flag if the function is started, 0 otherwise.

Notes:

When the function `start_cluster` is executed, a message is sent from the calling program to the appropriate cluster processor. The processor may or may not start the function it is told to execute depending on the value of `efnc`. However, after the message has been sent, the calling program continues its execution. Each parameter is assumed to be a memory descriptor returned by alloc_clmem. However, 30 bit integers may also be passed by encoding them using the param_const() function. (Two bits are used as a tag.) These constants will be sign-extended to 32 bits when actually used.

3.9.4 START_WARP

Start Warp machine after event flag is done.

Interface:

```
int start_warp(efnc)
EFLAG efnc;
```

Parameters:

`efnc`	An event flag to be set to DONE before the Warp array is started. In case there is nothing to wait for, `efnc` can be replaced with the constant NOWAIT.

Returns:

0 if the Warp is started, 1 otherwise.

3.9.5 CONTINUE_WARP

Continue the execution of the cluster processors and the Warp array after a breakpoint was encountered.

Interface:

```
int continue_warp(cycles)
int cycles;
```

Parameters:

`cycles`	How many cycles to run (-1 = until breakpoint or completion).

Returns:

0 if the execution completes, 1 otherwise.

3.9.6 EXECUTION_TIME

Return number of clock ticks elapsed during execution of last W2 program.

Interface:

```
int execution_time()
```

Parameters: None.

Returns: Return number of clock ticks elapsed during execution of last W2 program.

Notes: A clock tick is 100 microseconds. To get the time in milliseconds, divide by 10.

3.10 Debugging Functions

3.10.1 USE_PRINTF

When running standalone applications, it is possible to insert print statements of the form

```
printf("format string", p1, p2, ..., pn);
```

into cluster code programs. The function `use_printf` enables or disables these print statements when execution the application.

Interface:

```
int use_printf(flag)
int flag;
```

Parameters:

`flag`	if `flag` == 1, printing is enabled. If `flag` == 0, printing is disabled. By default, printing is disabled.

Returns:

0 if successful, 1 otherwise.

Notes: `openconn` must have been called before `use_printf` can be called. `use_printf` can only be issued in direct mode, that is, when running on a Warp host. It is a NOOP when running in remote mode. The implementation of `printf` is restricted: At most 8 arguments can be used and the format string must be a literal with less than 128 characters (that is, statements like

```
{ char *s = "Testing.\n"; printf(s); }
```

won't work). When printing is enabled, a SIGALRM is periodically generated and caught to poll the cluster processors. As a result, other uses of SIGALRM are prohibited. The cluster processors do not queue multiple print requests, so execution speed is largely controlled by the frequency of polling.

3.10.2 SET_BREAK

Set a breakpoint.

Interface:

```
int set_break(cell,cnt,brks)
int cell, cnt, brks[];
```

Parameters:

cell	Which cell to test (-1 = all Warp cells, 0 = IU).
cnt	How many breakpoints are to be set.
brks	An array of addresses of micro-instructions in which to set the breakpoint point bit.

Returns:

0 if successful, 1 otherwise.

3.10.3 CLEAR_BREAK

Delete a previously set breakpoint.

Interface:

```
int clear_break(cell,cnt,brks)
int cell, cnt, brks[];
```

Parameters:

cell	Which cell to test (-1 = all Warp cells, 0 = IU).
cnt	How many breakpoints are to be cleared.
brks	An array of addresses of micro-instructions in which to clear the breakpoint bit. (An entry of -1 means clear all breakpoints for this cell).

Returns:

0 if successful, 1 otherwise.

3.10.4 READ_DATA_MEM

Read data memory from one or more Warp cells.

Interface:

```
char *read_data_mem(cell,lo,hi,buf)
int cell, lo, hi;
char *buf;
```

Parameters:

cell	Which cell (or cells) to read from (0 = IU, n = cell n, -n = cells 1 through n).
lo	The lowest address to be read.
hi	The highest address to be read.
buf	A buffer to hold the data. If 0, a buffer will be allocated.

Returns:

0 if there was a problem with the read and a buffer pointer otherwise.

3.10.5 WRITE_DATA_MEM

Write data memory into one or more Warp cells.

Interface:

```
int write_data_mem(cell,lo,hi,buf)
int cell, lo, hi;
char *buf;
```

Parameters:

`cell`	Which cell (or cells) to write from (0 = IU, n = cell n, -n = cells 1 through n).
`lo`	The lowest address to be written.
`hi`	The highest address to be written.
`buf`	A buffer that holds the data to be written.

Returns:

0 if successful and 1 otherwise.

Notes:

In the case of multiple cells, both `read_data_mem` and `write_data_mem` assume that all the data for the first cell appears before any data from another cell.

3.10.6 SET_PC

Set the program counter for the Warp cells and the Interface Unit (IU).

Interface:

```
int set_pc(cnt,vector)
int cnt, vector[];
```

Parameters:

`cnt`	The number of cells to be used.
`vector`	An array of program counters for the IU and Warp cells.

Returns:

0 if successful and 1 otherwise.

Notes:

`vector[0]` is the IU program counter, and `vector[1]...vector[cnt]` are the program counters for the Warp cells. Microcode programs generated by the W2 compiler generally start at microcode address 100. Loading new microcode invalidates the result of this call.

3.10.7 GET_PC

Get the program counter for the Warp cells and the Interface Unit (IU).

Interface:

```
int get_pc(vector)
int vector[];
```

Parameters:

`vector`	A buffer to store program counters for the IU and all Warp cells.

Returns:

0 if successful and 1 otherwise.

Notes:

`vector` has to be declared as an array of 11 integers.

3.10.8 READ_MICROCODE

Read microcode from a Warp cell.

Interface:

```
char *read_microcode(cell, start, end, data)
int cell, start, end;
char *data;
```

Parameters:

`cell`	Where to read the microcode from (0 = IU, n = cell n).
`start`	First address to read.
`end`	Last address to read.
`data`	A buffer to store the microcode in. If 0, a buffer will be allocated.

Returns:

0 if there was a problem with the read and a buffer pointer otherwise.

3.10.9 WRITE_MICROCODE

Write microcode into a Warp cell.

Interface:

```
int write_microcode(cell, start, end, data)
int cell, start, end;
char *data;
```

Parameters:

`cell`	Where to write the microcode to (0 = IU, n = cell n).
`start`	First address to write.
`end`	Last address to write.
`data`	A buffer containing the microcode.

Returns:

0 if successful and 1 otherwise.

3.10.10 READ_CHAIN

Read from a serial chain.

Interface:

```
char *read_chain(cell, chain, buf)
int cell;
char *chain, *buf;
```

Parameters:

`cell`	What cell to examine.	
`chain`	Which chain to examine:	
	"cc"	IU control chain.
	"cd"	IU data chain.
	"ca"	IU address chain.
	"sc"	Warp control chain.
	"sd"	Warp data chain.
	"st"	Warp status chain.

	`buf`	A 40 byte buffer to store the chain in. If 0, a buffer will be allocated.

Returns:

0 if there was a problem with the read and a buffer pointer otherwise.

3.10.11 WRITE_CHAIN

Write to a serial chain.

Interface:

```
int write_chain(cell, chain, buf)
int cell;
char *chain, *buf;
```

Parameters:

`cell`	What cell to store into.	
`chain`	What chain to write into:	
	"cc"	IU control chain.
	"cd"	IU data chain.
	"ca"	IU address chain.
	"sc"	Warp control chain.
	"sd"	Warp data chain.
	"st"	Warp status chain.
`buf`	A 40 byte buffer containing the new chain data.	

Returns:

0 if successful and 1 otherwise.

3.10.12 READ_REGISTER

Read from register.

Interface:

```
int read_register(reg, val, op)
char *reg, *op;
int val;
```

Parameters:

`reg`	The register to be read. The names of the registers can be found in **$WARPserver/MONITOR.SRC/hardware.c**.
`val`	A bitmask.
`op`	An operation to be performed on the bitmask and the value read. (One of the following: "NOP", "OR", "AND", "XOR", "TAG" , "BYPASS", "CV16U", "CV16S", "CV8U", "CV8S")

Returns:

0 if there is a problem, otherwise the register value.

Notes:

The function `server_error` should be used to determine if a return value of 0 is an error or the value of the register.

3.10.13 WRITE_REGISTER

Write a register.

Interface:

```
int write_register(reg,val,op)
char *reg, *op;
int val;
```

Parameters:

`reg`	The register to be written. The names of the registers can be found in **$WARPserver/MONITOR.SRC/hardware.c**.
`val`	A bitmask.
`op`	An operation to be performed on the bitmask and the value read from the register. The result is written back to the register. (One of the following: "NOP", "OR", "AND", "XOR", "TAG", "BYPASS", "CV16U", "CV16S", "CV8U", "CV8S")

Returns:

0 if successful and 1 otherwise.

3.10.14 GET_FIELD

Get a field from a serial chain.

Interface:

```
int get_field(chain,field,buf)
char *chain, *field, buf[40];
```

Parameters:

`chain`	chain The name of the chain containing the field	
	"cc"	IU control chain.
	"cd"	IU data chain.
	"ca"	IU address chain.
	"sc"	Warp control chain.
	"sd"	Warp data chain.
	"st"	Warp status chain.
`field`	The name of the field to be extracted. The names of the fields can be found in **$WARPserver/include/chain.h**.	
`buf`	A buffer containing the chain data.	

Returns:

0 if there is a problem with the read and the value read otherwise.

Notes: The function `server_error` should be used to determine if a return value of 0 is an error or the value of the register. Only fields that are 32 bits or less can be retrieved.

3.10.15 PUT_FIELD

Put a value into a field of a serial chain.

Interface:

```
int put_field(chain,field,value,buf)
char *chain, *field, buf[40];
int value;
```

Parameters:

`chain`	The name of the chain containing the field	
	"cc"	IU control chain.
	"cd"	IU data chain.
	"ca"	IU address chain.
	"sc"	Warp control chain.
	"sd"	Warp data chain.
	"st"	Warp status chain.
`field`	The name of the field to be set. The names of the fields can be found in **$WARPserver/include/chain.h.**	
`value`	The new value of the field.	
`buf`	A buffer containing the chain data.	

Returns:

0 if successful and 1 otherwise.

Notes: This function does not modify the Warp machine, only the contents of the buffer. The function `write_chain` must be used to actually make the change. Only fields that are 32 bits or less can be set.

3.11 Miscellaneous Functions

3.11.1 SET_DEBUG

Trace message traffic between application and Warp/user server. The format of the messages is described in Section 4.3.

Interface:

```
set_debug(flag)
int flag;
```

Parameters:

`flag`

0 = Don't print any debugging information.

1 = Trace network traffic between application and user server.

2 = Announce whenever microcode is actually being loaded as a result of calling `load_micro` or `fast_load_micro`.

3 = Do both options, 1 and 2.

Returns:

None.

3.11.2 SET_TIMEOUT

Set length of time-out for the `execute_warp` call.

Interface:

```
set_timeout(secs)
int secs;
```

Parameters:

`secs` The time-out value in seconds. The default value is 15.

Returns:

0 if successful and 1 otherwise.

Notes:

`set_timeout` should be used if the Warp machine times out too early. Any time-out value less than 15 seconds defaults to 15.

3.11.3 GET_VERSION

Return the version number of the Warp monitor.

Interface:

```
get_version(buf, which)
char *buf;
int which;
```

Parameters:

`buf` A buffer that holds the version number.

`which`

0 = Version number of Warp monitor linked into the application.

1 = Version number of Warp monitor used by the user server.

Returns:

0 if successful and 1 otherwise.

Notes:

When running in local mode, the two version numbers are always identical, because the application runs without a user server. In remote mode, the version numbers should also be identical. Differing version numbers indicate that a new Warp monitor was installed after the application program was compiled and linked. In this case, it is recommended to recompile the application program.

3.11.4 PARAM_CONST

Create a constant that can be passed as parameter to `execute_warp` or `start_cluster`.

Interface:

```
int param_const(const)
int const;
```

Parameters:

`const` Constant to be encoded. Constant must be 30 bits long.

Returns:

a 32 bit integer where the two high order bit indicate this to be a constant.

3.11.5 SET_DIR

Change the current directory for file lookups done by the user server.

Interface:

```
int set_dir(dirname)
char *dirname;
```

Parameters:

`dirname` The absolute pathname of the directory that the server should use.

Returns:

0 if directory was changed, 1 otherwise.

Notes: `openconn` automatically performs a `set_dir` call to the current directory of the application.

3.11.6 SANITY_CHECK

Check the state of the file server, Warp host, the Warp server, WPE server and Warp array..

Interface:

```
int sanity_check()
```

Parameters: None.

Returns:

1	Cannot execute a remote shell call.
2	Can't connect to Warp host.
4	Cannot find Warp server.
8	Cannot find WPE server.
16	Cannot fork off user server.
32	Warp array is not accessible (Not able to find the IU).

Notes: If **sanity_check** finds multiple errors , then the above return codes are added together.

3.11.7 RESET_WARP

Reset the Warp machine to a defined state.

Interface:

```
int reset_warp()
```

Parameters: None.

Returns: 0 if the machine has been reset and 1 otherwise.

Notes: This function is the same as running **$WARPbin/reset_warp**.

3.12 Using the Warp Monitor

This section contains examples of standalone application programs. First we give a detailed discussion of two C programs using the Warp monitor. We will explain the use of the Warp monitor functions and how to compile, link and execute the programs. Then we show how the Warp monitor can be made available to the Common Lisp programmer.

3.12.1 Pipe I: Using the Warp Monitor inside a C Program

The basic task of the C program in figure 3-2 is to create ten input data, call a Warp program `pipe` which adds the value `1` to each input value and to print the result on the terminal. The Warp program is located in directory **$WARPserver/W2/PIPE/**.

Line 2 includes the file **monitor.h** which contains constant and variable definitions needed by the Warp monitor.

Line 7 declares the variables `parin` and `parout` that will contain the memory descriptors of the input and output parameters of the Warp Program. Line 8 declares a variable `data` as a pointer to a float. `data` holds a buffer pointer returned by `get_uninit_clmem` and `read_from_clmem`. In line 10 the Warp monitor function `openconn` is called. The first parameter is the name of a Warp host and must passed from the C-shell to the C program. The second parameter selects the server type 3: If the C application is in remote mode, all Warp monitor

calls are remote procedure calls, if it is in local mode, all Warp monitor calls are done by direct procedure calls. The call `set_debug` in line 11 turns on tracing of the message traffic between Warp server and the C program (This call can be omitted or changed into `set_debug(0)` after the program is debugged). Line 12 tells the user server the directory for looking up the micro and cluster code for the pipe program.

In line 13 the Warp machine is locked. In lines 14 to 15 cluster memory is allocated for the input and output parameter of the Warp program. If the memory allocation was successful, then `parin[0]` and `parout[0]` contain memory descriptors for the allocated cluster memory areas. The call `get_uninit_clmem` in line 16 returns a buffer for the cluster memory described by `parin[0]`. Lines 17 to 19 initialize `data` and in line 20 the contents of `data` is copied into the cluster memory described by memory descriptor `parin[0].free_buffer` in line 22 disposes of the input data pointer because it is no longer needed. Lines 23 to 24 load the micro code and cluster code which will looked up in directory `"$WARPserver/W2/PIPE/"` in our example.

In Line 25 the Warp program is called. Line 26 reads the result from the cluster memory described by memory descriptor `parout[0]` into the variable `data`. Lines 27 to 28 print the result on the terminal. In line 29 the Warp machine is released.

```
#include <stdio.h>                                                                  /* 1*/
#include <monitor.h>                                                                /* 2*/
  extern char *getenv();                                                            /* 3*/
  main(argc, argv)                                                                  /* 4*/
int argc;  char ** argv;                                                            /* 5*/
    {  char          *exdir; char pipedir[256];                                     /* 6*/
       int            parin[1], parout[1], i;                                      /* 7*/
       float         *data;                                                        /* 8*/
     openconn(argv[1], 3);                                                          /* 9*/
     set_debug(1);                                                                  /*10*/
     exdir = getenv("WPEexamples"); sprintf(pipedir,"%s%s",exdir,"w2");             /*11*/
     set_dir(pipedir);                                                              /*12*/
     if (lock_warp(1, 1, "Testing") == 1) exit(0);                                  /*13*/
     parin[0] = alloc_clmem("dk", 10 * sizeof(float));                              /*14*/
     parout[0] = alloc_clmem("dk", 10 * sizeof(float));                             /*15*/
     data = (float *) get_uninit_clmem(parin[0], 0, 10 * sizeof(float));            /*16*/
     for (i = 0; i < 10; i++) data[i] = i;                                          /*17*/
     for (i = 0; i < 10; i++) printf("%4.1f ", data[i]);                            /*18*/
     printf("\n");                                                                  /*19*/
     if (write_to_clmem(parin[0], 0, 10 * sizeof(float), data))                     /*20*/
        exit(0);                                                                    /*21*/
     free_buffer(data);                                                             /*22*/
     load_micro("pipe");                                                            /*23*/
     load_cluster("pipe");                                                          /*24*/
     if (execute_warp(-1, 1, 1, parin, parout) < 0) exit(0);                        /*25*/
     data = (float *) read_from_clmem(parout[0], 0, 10 * sizeof(float), 0);         /*26*/
     for (i = 0; i < 10; i++) printf("%4.1f ", data[i]);                            /*27*/
     printf("\n");                                                                  /*28*/
     unlock_warp(1);                                                                /*29*/
  }                                                                                 /*30*/
```

Figure 3-2: A C Program Calling a W2 Program

3.12.2 Pipe II: Using Event Flags Functions

This section contains an example of an application using event flag functions. The program in figure 3-3 basically performs the same task as in figure 3-2. The main difference is that the cluster processors and the Warp array are now treated as separate processors. Functions are explicitly started on each of these components and event flags are used to synchronize their execution. In the following we discuss only those parts of the program that are different from the previous example.

Line 3 includes the file **Wconfig.h** which contains constant and variable definitions needed by the event flag

```
                                                                        /* 1*/
#include <stdio.h>                                                      /* 2*/
#include <monitor.h>                                                    /* 3*/
#include <Wconfig.h>                                                    /* 4*/
#define SIZE 10                                                         /* 5*/
 extern char *getenv();                                                 /* 6*/
 main(argc, argv)                                                       /* 7*/
  int argc;                                                             /* 8*/
  char ** argv;                                                         /* 9*/
   {    char            *exdir;                                        /*10*/
        char             pipedir[256];                                 /*11*/
        int              parin[1], parout[1], i;                       /*12*/
        float           *data;                                         /*13*/
        EFLAG            ef, ef1, ef2;                                 /*14*/
        int              mid;                                          /*15*/
        int              func1, func2;                                 /*16*/
        openconn(argv[1], 3);                                          /*17*/
        exdir = getenv("WPEexamples");                                 /*18*/
        sprintf(pipedir,"%s%s",exdir,"w2");                            /*19*/
        set_dir(pipedir);                                              /*20*/
        if (lock_warp(1, 1, "Testing") == 1) exit(0);                  /*21*/
        parin[0] = alloc_clmem("C1M3", SIZE * sizeof(float));          /*22*/
        parout[0] = alloc_clmem("C2M2", SIZE * sizeof(float));         /*23*/
        mid = cache_micro("pipe");                                     /*24*/
        ef = fast_load_micro(mid, NOWAIT);                             /*25*/
        func1 = make_clust_func("pipein.mem");                         /*26*/
        func2 = make_clust_func("pipeout.mem");                        /*27*/
        data = (float *) get_uninit_clmem(parin[0], 0, SIZE * sizeof(float));  /*28*/
        for (i = 0; i < SIZE; i++) data[i] = i;                        /*29*/
        printf("\nInput:\n");                                          /*30*/
        for (i = 0; i < SIZE; i++) printf("%5.1f ", data[i]); printf("\n");  /*31*/
        if (write_to_clmem(parin[0], 0, SIZE * sizeof(float), data)) exit(0);  /*32*/
        free_buffer(data);                                             /*33*/
        start_warp(ef);                                                /*34*/
        ef1 = start_cluster(CLUSTER1, func1, ef, parin[0], -1);        /*35*/
        ef2 = start_cluster(CLUSTER2, func2, ef, parout[0], -1);       /*36*/
        wait_ef(ef2, DONE, 0);                                         /*37*/
        free_ef(ef1); free_ef(ef2); free_ef(ef);                       /*38*/
        data = (float *) read_from_clmem(parout[0], 0, SIZE * sizeof(float), 0);  /*39*/
        printf("\nOutput:\n");                                         /*40*/
        for (i = 0; i < SIZE; i++) printf("%5.1f ", data[i]); printf("\n");  /*41*/
        unlock_warp(1);                                                /*42*/
   }
```

Figure 3-3: A C Program Using Event Flag Functions

functions. Line 11 defines three event flag variables, line 12 a microcode ID variable, and line 13 two cluster code function ID's. Lines 23 and 24 load the micro code. The primary use of `cache_micro` and `fast_load_micro` is to cache microcode into Warp host memory so it can be loaded more quickly into the Warp array. This is of advantage for real-time application that switches between several Warp programs. If the micro code has to be loaded only once, the `load_micro` function of the previous example is sufficient. Lines 25 to 26 load two cluster code functions into cluster processor memory. Two function code ID's are returned, which are stored in the variables `func1` and `func2`, respectively.

Lines 33 to 35 are equivalent to the function call `execute_warp` in line 25 of the example in figure 3-2. Line 33 starts the Warp array, line 34 starts function `func1` in cluster processor CLUSTER1 and line 35 starts `func2` in CLUSTER2. All of these functions wait for the event flag `ef` to be set to DONE before they start their execution. Line 36 waits for event flag `ef2` to be set to DONE, which means the output result of the Warp program is available and can be printed. Line 37 deallocates the event flags `ef`, `ef1` and `ef2`.

3.12.3 Compiling, Linking and Executing C Programs

The Warp monitor is implemented such that there is no difference between the core image of an application running in remote mode or in local mode. Thus, application programmers can compile, link and test their applications in the familiar environment of their personal workstation before they download them to the Warp host for local execution. Let us assume the C programs from Section 3.12.1 and 3.12.2 are called `test.c` and `testef.c`, respectively. The following Makefile compiles and links them into runfiles `test` and `testef`. Compiling these files requires access to the directory **$WARPlib/**. In addition, the compilation of `testef` requires access to the directory **$WARPinclude/** which contains the definition file **Wconfig.h** for the event flag functions:

```
LIBS = $WARPlib/master.a $WARPlib/warplib.a

test: test.c $WARPlib/monitor.a
        cc -o test test.c $WARPlib/monitor.a $(LIBS)
testef: testef.c $WARPlib/monitor.a
        cc -o testef testef.c -I$WARPinclude/ \
        $WARPlib/monitor.a $(LIBS)

clean:
        -rm Makefile.doc
        -rm test
        -rm testef
        -rm *.err
        -rm *.otl
        -rm *~
```

Let us assume the runfile `test` resides in directory **/usr/bob/wpe/test/C/** and we want to execute it on Warp host `warpm`. Typing the command

```
% /usr/bob/wpe/test/C/test warpm
```

to the Warp shell on the workstation `storch` generates the following output:

```
>>> 1000 bob storch /tmp/E_IPC_3145 "" 1
<<< 2000 "Connection Accepted"
>>> 1004 /usr/pcwarp/support/server/W2/PIPE/
<<< 2004 "Directory Changed"
>>> 1001 -1 1 "Testing"
<<< 2001 1 "Lock Granted"
>>> 1005 dk 40
<<< 2005 0 "Memory Allocated"
>>> 1005 dk 40
<<< 2005 1 "Memory Allocated"
>>> 1007 0 0 40 "" U
<<< 2007 "Read Done"
 0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0
>>> 1008 0 0 40 "" C
<<< 2008 "Write Done"
>>> 1009 pipe
<<< 2009 "Microcode Loaded"
>>> 1010 pipe
<<< 2010 "Cluster Code Loaded"
>>> 1011 -1 1 1 0 1
<<< 2011 "Execution Completed" 0
>>> 1007 1 0 40 "" C
<<< 2007 "Read Done"
 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
>>> 1002
<<< 2002 "Warp Released"
%
```

Executing **testef** yields the following output on the terminal:

```
Input:
  0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0

Output:
  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0
```

3.12.4 Using the Warp Monitor inside Common Lisp Programs

The Warp monitor is also accessible for Lucid Common Lisp users. The Lisp interface to the Warp monitor resides in the file `$WPEroot/warpshell/w2rpc.slisp` and is defined in the package `Shell`. If you are running the Warp shell, this file is already loaded by default. If you are running a standalone application, the Warp monitor can be loaded with the following Common Lisp command[35]:

```
(load "$WPEroot/warpshell/w2rpc.slisp")
```

The function `Shell::DefineServerFunctions` defines the Common Lisp interface to the C functions. For example, the line

```
(lucid::define-c-function load_micro (file) :result-type :integer)
```

makes the Warp monitor function `load_micro` accessible within the package `Shell` and can be called like a normally defined Lisp function. For example, the command

```
(setf returncode (load_micro "/usr/bob/test/pipe/pipe"))
```

downloads the microcode file **pipe.w1** stored in directory **/usr/bob/test/pipe/** to the Warp machine.

[35] **$WPEroot** has to be replaced by its value.

4. When Things Go Wrong

4.1 Known Bugs

1. TMP-BUG: Currently it is not possible to allocate Warp shell variables on the **/tmp** file structure. The reason is that two different /tmp's exist: **/tmp** on your workstation and **/tmp** on the Warp host. The Warp shell evaluates all file names with respect to your workstation, and the user servers evaluate all filenames with respect to the file structure used by the Warp host. Most of the file structures such **/usr/** are crossmounted and should not be a problem. However, **/tmp/** is not crossmounted. For the current time all file names should therefore be located in structure **/usr/**. BUG-FIX: A machine-wide file naming system will fix this problem.

2. COMPOSITE C SHELL COMMANDS: The pipe command (|) does not work for C-shell commands.

3. Bugs when using GNU Emacs:

   - GOTO-SHELL-BUG: The GNU Emacs command `CTL X CTL W (write-file)` used to change the name of the buffer to the name of the file being written. This has been changed in WPE 2.4: The buffer name is not changed.

4. Bugs when using Gosling's Emacs:

   - PR-DOT-BUG: Every now and then Gosling's Emacs gets confused which buffer to use for the current output. In this case you will get the message "progn variable pr-dot has been unbound" or "process output with no destination". Common to all appearances of the PR-DOT-BUG is that everything seems to be frozen, but as soon as you move the cursor with CTL N or CTL P, you get one of the above error messages. Please send mail to the system maintainer you run into this problem. BUG-FIX: The command W2-SET -SHOWCOMMANDFILE OFF usually minimizes the occurrence of the bug and is therefore currently automatically executed by `$WPEhome/warpshellinit.cmd`.

5. SU AND LOGIN: When using the UNIX shell commands SU (superuser) or LOGIN, the password is echoed. And it is not possible to return to the Warp shell afterwards.

4.2 Error Messages

This section contains error messages issued by the WPE components. The list of error messages is by no means exhaustive. We have selected those error messages that seem to be especially obscure for the first-time or casual WPE user. Please send mail to the WPE maintainer, if you experience other error messages that deserve to be included here.

Problems with $WPEbin/wpecoreimage

If you start too many WPE's on one workstation, you get the GNU Emacs error message

```
Couldn't exec the program $WPEbin/wpecoreimage
Process lisp exited abnormally with code 1
```

or the Gosling Emacs error message

```
$WPEbin/wpecoreimage: Not enough core
Exited Abnormally
```

Usually this occurs when an obsolete WPE core image has not been killed properly. Type `ps aux | grep wpecoreimage` to determine the process id of the associated UNIX process and kill it with `kill -9`.

Symbol's function definition is void

If you get the error message:

```
Error in init file: Symbol's function definition is void: wsh
```

then the boot file **warpshell.el** is not accessible. Check your ~/.emacs file, whether **load-path** is defined such that it looks up files in the directory **$WPEeditor/**.

Not enough memory

The error messsage

```
not enough memory!!!
```

might occur during the compilation of a W2 program. It means that there are too many UNIX processes running on your workstation while one of the W2 compiler utilities is trying to allocate memory. Kill some processes and retry the compilation. If there are still problems, try to compile with a different -spicy option (see W2-COMPILE, section 2.2.8, page 16).

Lisp Errors

Messages of the form

```
>>Error: ....
```

are generated by a call to the Common Lisp function `error`. Many functions in the W2 compiler call `error` when they encounter an unrecoverable error. Usually these error messages are self-explanatory. For example, if you try to compile a nonexisting W2 program **foo.w2**, you will get the error mesage

```
>>Error: the file foo.w2 does not exist
```

If you are a "developer" or "user", the Warp shell will return back to the prompt. If you are an "implementor", the Warp shell will invoke the Common Lisp debugger.

XIO: Broken pipe

If you telnet to a remote workstation and you invoke WPE without having authorized the remote workstation to invoke GNU Emacs on your local workstation, then you get the error message

```
XIO: Broken pipe
```

Set the environment DISPLAY on the remote workstation to the name of your workstation and run **xhost** so the remote workstation can display on your workstation.

Could not copy variable into cluster memory

If you try to initialize a Warp shell variable, you might get the following dialog:

```
% w2-type float10 array[10] of float
Type "float10" created
% w2-var -name input -type float10 -init -textfile indata
Allocating memory for "input" ...initializing...
? Could not copy variable "input" into cluster memory
Variable "input" created
```

Note that the variable has been created. The error message means that the variable was not initialized. There are several possible reasons: The file might contain too few data. Maybe it is empty or it does not exist. Or the file is in a protected directory.

gcc: Command not found

This error message might occur, when WPE is running in remote mode and you are trying to compile a W2 program. The problem is usually that your workstation does not have access to the Green Hills C compiler. The Green Hills C compiler is invoked by the W2-COMPILE command to generate the code for the cluster processors.

Invoke W2-COMPILE with the **-remote** option and specify the name of the Warp host (see Section 2.2.8, page 16). There is always a Green Hills compiler on the Warp host.

Unexpected end-of-file

```
>>Error: Unexpected end-of-file encountered during read.

READ:
   Optional arg 0 (STREAM): #<Stream BUFFERED-STREAM 86101B>
   Optional arg 1 (EOF-ERROR-P): T
   Optional arg 2 (EOF-VALUE): NIL
   Optional arg 3 (RECURSIVE-P): NIL

:A    Abort to Lisp Top Level
->
```

This error message occurs, if you load a Warp shell command file with multiple lines of Common Lisp code where the individual lines are not terminated by semicolons (see Section 2.2.2).

Lisp process finished

If you get the message

```
Lisp process finished
```

right after invoking WPE, then the run file **wpecoreimage** found by the **wpeg** command is not a lisp core image. Check out whether you have a bad core image hanging around somewhere.

Problem Sending Command to Emacs

WPE communicates with EMACS with the help of a file whose name is of the form "`E_IPC_pid=`", which is allocated on the **/tmp/** file structure when WPE is started up (`pid` is the UNIX process id number of WPE).

Error messages of the form

```
? Problem sending command "(foo)" to Emacs
```

occur if you have deleted this file while WPE is running. This can happen if you clean **/tmp** too thoroughly.

GIL Error Messages

Error messages from the generalized image library are prefixed with the name of the routine (usually of the form GIL: i_<foo>) followed by a short description of the error and an interrupt which causes a segmentation violation in the Warp shell. Type **:a** to return to the command interpreter. The following is are typical GIL error messages:

i_net: Read error on network. Connection timed out
: You try to read an image over the network and the other site is down.

i_malloc: Warning: Called from IM_strcpy for 8022 bytes
: Error messages of this form are used for debugging the GIL in its use of memory. They can be ignored.

Could not open X display

If you are trying to create an X window on a workstation specified by the UNIX environment variable DISPLAY you must have access rights to this display. An error message like

```
GIL: i_xwincreat: Could not open X display '(default)'
```

occurs when you try to display an image on an X window and you don't have any display rights on that workstation. You have to run **xhost** to add the workstation to the list of workstations allowed to open connections.

4.3 Monitoring The Warp and Users Servers

The Warp and user servers manage the Warp machine in a location-transparent way. Ideally, they should not be visible to the user. Of course, sometimes this does not work. A problem occurs, for instance, if you have locked the Warp machine and the Warp server crashes. Or the Warp server might be down when you try to lock the Warp machine. In this case you will get the message

```
?User server: User server died.
```

You can use the W2-SANITY command to check whether the Warp host or the Warp server is causing the problem.

Every now and then there will be less obvious problems. For example, the user server might tell you that it cannot allocate memory in the cluster processor memory, cannot read a file, cannot load micro code, or you might get a message that the file system is full. In this case, turn on a tracing facility with the command W2-TRACE -SERVER ON that allows you to watch the message traffic between the Warp shell and the user server. Messages prefixed by a >>> indicate a message from the Warp shell to the user server, and messages prefixed by a <<< indicate a message from the user server to the Warp shell. A summary of all error codes used by the Warp shell is described below. The message tracing facility is enabled until you issue the command W2-TRACE -SERVER OFF.

All messages to and from the user server are lines of text terminated with a newline. Each line consists of a number of fields separated by spaces. (Spaces may be included in a field by quoting the entire field in double quotes ("")). The first field is always a four digit number that identifies the message. The most significant digit of this number is the type of the message: 1 = Server command, 2 = command completed, 3 = command failed, 4 = fatal command failure. The last three digits identify the command being performed. So for command N, 1000+N is the client request, 2000+N is the success indicator, and 3000+N is the failure indicator. The possible commands are as follows:

```
 0 - Initialize connection to Warp server.
 1 - Lock the Warp machine.
 2 - Unlock the Warp machine.
 3 - List the jobs in the queue.
 4 - Change the current directory.
 5 - Allocate memory.
 6 - Deallocate memory.
 7 - Read into memory allocated by command 5.
 8 - Write from memory allocated by command 5.
 9 - Load Warp microcode from a file.
10 - Load cluster code from a file.
11 - Start running the cluster processors and the Warp machine.
12 - Continue running the Warp machine (after a breakpoint).
13 - Read into Warp cell data memory.
14 - Write from Warp cell data memory.
15 - Set cell program counters.
16 - Read cell program counters.
17 - Set breakpoint(s).
18 - Clear breakpoint(s).
19 - Read into microcode memory.
20 - Write from microcode memory.
21 - Read serial chain.
22 - Write serial chain.
23 - Read from register.
24 - Write to register.
25 - Reset Warp machine.
26 - Allocate event flag.
27 - Set event flag.
28 - Wait for event flag.
29 - Cache microcode.
30 - Make cluster function.
31 - Load cached microcode.
32 - Start cluster function.
33 - Start Warp machine.
34 - Copy cluster to cluster.
```

The following is an example of a message trace created during the successful execution of a w2 program:

```
% w2-trace -server on
Message traffic between Warp server and Warp shell will be traced
% w2-execute -file /usr/bob/wpe/test/pipe/pipe -parameters input output
Copying input parameters into cluster memory...
>>> 1004 /usrw61/bob/wpe/test/pipe/
<<< 2004 "Directory changed."
Downloading microcode for module "pipe" from directory "/usrw61/bob/wpe/test/pipe/"
Locking Warp GE...
>>> 1003
<<< 2003 0 "Queue Follows."
<<< .
Warp server queue is empty.
>>> 1001 -1 1 "Running Warp Programming Environment"
<<< 2001 "Lock Granted."
Warp machine is yours:
>>> 1009 pipe
<<< 2009 pipe "Loaded."
>>> 1010 pipe
<<< 2010 pipe "Loaded."
>>> 1015 1 100 100
<<< 2015 "PC set."
Starting execution of module...
>>> 1011 -1 1 1 0 1
<<< 2011 1 "Execution complete."
Execution completed.
Copying output parameters from cluster memory...
Writing "output"  into file "/usr/bob/wpe/test/pipe/outdata"...
>>> 1008 1 0 40 /usr/bob/wpe/test/pipe/outdata float
<<< 2008 "Elements written."
Unlocking Warp GE...
>>> 1002
<<< 2002 "Lock Released."
%
```

4.4 Creating Bug Reports

If you have found a bug in the Warp Shell, do the following steps:

1. Execute Warp shell command W2-SET -USER IMPLEMENTOR (This will invoke the Common Lisp debugger when the bug is repeated).
2. Repeat the last command and wait until the prompt of the Common Lisp debugger appears.
3. Get a trace of the call stack by typing **:b** to the Lisp debugger.
4. Save the session into a file and send it with other important files to the maintainer of the Warp Programming Environment[36].

If you suspect a bug in the W2 compiler, do the above steps and in addition

5. Save the buffer `error-log` or `*compilation*` and mail it as well.

[36] If you are a CMU user, send the bug report to wpe@sam.cs.cmu.edu.

I. Summary of Warp Shell Commands

Any UNIX shell command is known by the Warp shell. In addition, the following commands are implemented:

```
alias             Define a new alias
copy              Copy one or more files
date              Print the current date and time
defcommand        Define a new command (as a Lisp function)
delete            Delete files
directory         List files that match a pattern
echo              Echo arguments
edit              Edit a file in an editor window
find              Find all occurrences of a file name
grep              Search a file for a pattern
handle-errors     Handle errors that occur within commands
help              Print this text, or help about a command or alias
history           Print recently executed commands
keyword           Print all commands matching a given key
load              Load a lisp file
make              Maintain related sets of programs
Noisy             Control the verbosity level of shell commands when they are executed
path              Change the current directory
Pause             Pause the shell
popdir            Change the current directory to the one on top of the stack
pushdir           Move to a new, or previous, directory
Quit              Quit the shell
rename            Rename or move one or more files
searchpath        Show the current search path
setsearch         Set the default search path
show              Show the value of an environment variable
sort              Sort lines from standard input
time              Time the execution of a Shell command
touch             Change the creation date of a file
type              Type out a file
undefcommand      Remove an alias or a command (in that order)
verbose           Print expanded version of commands before executing them
version           Current Shell version
w2-break          Set a source line break point
w2-compile        Compile a W2 program
w2-continue       Continue current W2 program
w2-delete         Delete a Warp shell object
w2-disable        Disable a breakpoint
w2-download       Download micro code for W2 program onto Warp array
w2-edit           Edit a user defined Warp shell variable
w2-enable         Enable a breakpoint
w2-execute        Execute w2 module
w2-get            Print all locals of the current function
w2-halt           Halt the Warp machine
w2-init           Initialize a WPE component
w2-load           Load a WPE component
w2-lockwarp       Reserve the Warp machine
w2-printnodes     Pretty print the dags of the flow graph
w2-reset          Reset the W2 debugger or the Warp server to its initial state
w2-restart        Restart current W2 program
w2-sanity         Check state of file server, Warp host, Warp server, WPE server and Warp.
w2-set            Set a variable of the current environment of the debugger
w2-show           Show the current environment
w2-suggestbreaks  Suggest possible breakpoints
w2-targetToSource Get all w2 source lines  for given w1 address
W2-trace          Turn on or off tracing information for Warp server and editor
w2-type           Declare a Warp shell type
w2-unlockwarp     Release the Warp machine
w2-var            Declare a Warp shell variable
w2-warpqueue      Show the users currently owning the Warp machine or waiting for it
wpeversion        Print the version number of WPE
```

II. Summary of Warp Monitor Functions

The following functions are currently implemented:

alloc_ef	Allocate event flag
alloc_clmem	Allocate memory in cluster memory
clear_break	Delete a breakpoint
cache_micro	Cache micro code file into Warp host memory
continue_warp	Continue execution
copy_ctc	Copy cluster memory to cluster memory after event flag is done
copy_ctf	Copy cluster memory into a file
copy_ftc	Copy a file into cluster memory
dealloc_clmem	Deallocate memory in cluster memory
execute_warp	Start the cluster processors and the Warp machine
fast_load_micro	Download cached micro code after event flag is done
free_buffer	Free memory allocated by read_from_clmem or get-uninit_clmem
free_ef	Free event flag
get_error_string	Get the error string from the last Warp monitor call
get_field	Get a field from a serial chain
get_pc	Get program counter
get_uninit_clmem	Get uninitialized cluster memory
get_version	Return the version number of the Warp monitor
list_queue	Return the number of jobs in the Warp server queue
load_cluster	Load cluster code for the two cluster processors
load_one_cluster	Load cluster code for one cluster processor
load_micro	Load Warp microcode into interface unit and Warp array
lock_warp	Lock the Warp machine
make_clust_func	Load cluster code, create function and return function code ID
next_entry	Get next entry from Warp server queue
openconn	Connect to a Warp host and select a Warp server type
put_field	Put a value into a field of a serial chain
param_const	Create constant parameter for execute_warp or start_cluster
read_chain	Read from a serial chain
read_data_mem	Read from Warp cell data memory
read_ef	Read event flag
read_from_clmem	Copy from cluster memory into a buffer
read_microcode	Read microcode from a Warp cell
read_register	Read from register
reset_warp	Reset the Warp machine to a defined state
sanity_check	Check state of file server, Warp host, Warp server
server_error	Return error index of last Warp monitor call
set_break	Set a breakpoint
set_debug	Trace message traffic between application and Warp/user server
set_dir	Change current directory for file lookups done by user server
set_ef	Set event flag
set_pc	Set program counter
set_timeout	Set time-out for execute_warp call
start_cluster	Start cluster processor given a list of parameters after event flag is done
start_cluster_a	Start cluster processor given an array of parameters after event flag is done.
start_warp	Start the Warp machine after event flag is done
start_warpd	Start the Warp server (demon)
steps_done	Number of steps executed
unlock_warp	Unlock the Warp machine
use_printf	Enable or disable printf statements in cluster code programs
wait_ef	Wait for event flag
write_chain	Write to a serial chain
write_data_mem	Write to Warp cell data memory
write_microcode	Write microcode
write_register	Write to register
write_to_clmem	Copy from a buffer to cluster memory

Topical Index

Warp Monitor Functions

Warp Shell Functions