2 Invoking the Debugger

iwdb Command

To invoke the debugger, use the iwdb command as follows:

`iwdb [`*`option`*` ...]`

option can be any of the options listed in the following section.

iwdb Options

`-batch`	Run in batch mode. Exit with code 0 after processing the command files specified with `-x` and `.iwdbinit`, if not inhibited. Exit with nonzero status if an error occurs in executing the debugger commands in the command files.
`-C` *configpath*	Specifies *configpath* as the path name to search for the configuration file. The default path is /iwarp/etc/cfg.
`-d` *directory*	Add *directory* to the path to search for source files.
`-H` *nodename*	Identify *nodename* as the array server network name. *nodename* must be a valid network machine name and correspond to an entry in the iWarp configuration file. The default is the local machine on which the debugger is running.
`-nx`	Do not execute commands from the init file .iwdbinit. Normally, the init file commands are executed after all command options and arguments have been processed.
`-pm`	Specifies post-mortem debugging. Refer to the section *Post-Mortem Debugging* in Chapter 3 for a description of post-mortem debugging.
`-q`	Do not print introductory messages.
`-S` *sibnum*	Use the SIB defined by *sibnum. sibnum* must correspond to an entry in the iWarp configuration file. The default value for *sibnum* is 0.
`-x` *file*	Execute debugger commands from *file*.

The following example shows the use of the `iwdb` command:

```
iwdb -S 1  -q
```

This example specifies the use of SIB number 1, and suppresses the printing of introductory debugger messages. Since the example does not specify an array server network name, the local machine on which the debugger is running would be selected by default.

Cell Context

In normal debugging, the debugger is concerned with one program at a time. In parallel debugging, there may be as many programs as cells in the array. Therefore, simply naming the program is not sufficient. The cell or cells where the program will run must also be specified. The iWarp debugger provides cell context commands to associate a program name with a list of cells.

Since the number of cells may be large and many cells may be running the same program, the cell context can be defined as a group. This associates a symbolic name (the group) with a list of cell numbers. The group name must be used to specify multiple cell numbers to the `context` command.

Cell Context Commands

The following cell context commands are available with the iWarp debugger:

`group` *`groupname cell_id1`* `[`*`cell_id2 . . .`* `]`

Associates *groupname* with the list of cell numbers as given. If *groupname* is already being used, a warning message will be issued asking for permission to change the group definition. Any response other than *y* or Y aborts the request. Cells are not restricted to only one context. The list of cell numbers is expanded and verified to lie within the range of the array in use. If any cell number is outside that range, an error message is issued and the request is ignored.

`add_group` *`groupname cell_id`* `[`*`cell_id2...`*`]`

Add one or more cell IDs to group *groupname*.

`context {`*`cell_id`* `|` *`groupname`*`}`

Establishes the cell *cell_id*, or the group *groupname*, as the current working context. All subsequent debugger commands will be relative to the established context until the context is changed.

`load` *`program`*

Loads *program* onto all cells of the current context. If there is no declared context, an error message is issued.

`cell-numbering` *`filename`*

Allows you to use a different cell numbering scheme. The default is left to right, top down, starting with zero (canonical). *filename* is a text file that maps the canonical number to a text string. Each line corresponds to one cell in the array. The cell numbering file has the following format:

```
!n,m,b,c
#num,string
#num,string
   •
   •
   •
$string,num
   •
   •
   •
.end
```

The specifiers are defined as follows:

n	the array width as defined in the configuration file
m	the array height as defined in the configuration file

b	the number of cell numbers reserved for auxiliary cells
c	the number of cell numbers used by auxiliary cells
num	a system number conforming to the ASCII number convention for system numbers, including leading zeros
string	any number string, excluding spaces, newlines, and commas

The following is an example of portions of a cell numbering file for an 8X8 array:

```
!8,8,2,2
#00,0X0
#01,0X1
#02,0X2
#03,0X3
#04,0X4
#05,0X5
#06,0X6
#07,0X7
#08,1X0
#09,1X1
#10,1X2
    •
    •
    •
#62,7X6
#63,7X7
#64,SIB
#65,HOST
$0X0,00
$0X1,01
$0X2,02
$0X3,03
$0X4,04
$0X5,05
$0X6,06
$0X7,07
$1X0,08
$1X1,09
$1X2,10
    •
    •
    •
$7X7,63
$SIB,64
$HOST,65
.end
```

`info group`

Lists the group names, followed by associated cellids for all defined groups.

`info context`

Lists the program name, group name (if any) and cellids of the current context. As a side-affect, the symbol table for the program is stored internally.

`info cell-numbering`

Lists the cell numbering currently in effect.

`info status`

Lists the status of the cells in the array: running, exited, or idle. Note that idle can mean either the program on the cell is paused at a break-point or that no program has been loaded. All output that pertains to a given context is preceded by the cell id or context name.

Specifying Files

Files are usually specified by giving arguments when you invoke the debugger. It is sometimes necessary to change to a different file during a debugger session, however. The debugger includes the following file specification commands to specify files during the debugger session.

File Specification Commands

`exec-file` *`filename`*

Specifies that the program to be run is found in *filename*. If you do not specify a directory and the file is not found in the debugger's working directory, the debugger will use the environment variable PATH as a list of directories to search, just as the shell does when looking for a program to run.

`symbol-file` *`filename`*

Reads symbol table information from *filename*. PATH is searched when necessary. The `exec-file` and `symbol-file` commands are commonly used on the same file.

`symbol-file` with no argument clears out the debugger's symbol table.

`info files`

Prints the names of the files currently in use by the debugger and the file from which symbols were loaded.

While all three file-specifying commands allow both absolute and relative file names as arguments, the debugger always converts the file name to an absolute file name and remembers it that way.

The `symbol-file` command causes the debugger to forget the contents of its convenience variables, value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside the debugger. Refer to Chapter 5 for a description of convenience variables and value history.

Command Sequences

The debugger provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

User-Defined Commands

A user-defined command is a sequence of debugger commands to which you assign a new command name. This is done with the `define` command. The `define` command has the following syntax:

```
define commandname
```

This command defines a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

The definition of the command consists of debugger command lines, which follow the `define` command. The end of these commands is marked by a line containing `end`.

You can also create on-line documentation for a user-defined command with the `document` command. The `document` command has the following syntax:

```
document commandname
```

This command creates documentation for the user-defined *commandname*. *commandname* must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, terminating with `end`. After the `document` command is finished, `help` *commandname* will print the documentation you have specified.

You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

User-defined commands do not take arguments. When they are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

Commands that would ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many debugger commands that normally print messages omit these messages when used in user-defined commands.

Command Files

A debugger command file is a file of debugger command lines. Comments (lines starting with `#`) can also be included. An empty line in a command file does nothing. It does not repeat the last command, as it would from the terminal.

When the debugger starts, it automatically executes its init files, which are command files named `.iwdbinit`. The debugger reads the init file (if any) in your home directory and then the init file (if any) in the current working directory. The init files are not executed if you use the `-nx` option.

You can also request the execution of a command file with the `source` command, as shown in the following example:

```
source filename
```

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that ask for confirmation if used interactively proceed without asking when used in a command file. Many debugger commands that normally print messages omit these messages when used in command files.

Commands for Controlled Output

During the execution of a command file or a user-defined command, the only output that appears is what is explicitly printed by the commands of the definition. The following commands are useful for generating the output you want.

`echo` *`text`*

Prints *text*. Nonprinting characters can be included in *text* using C escape sequences, such as `\n` to print a newline. No newline will be printed unless you specify one. In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for outputting a string with spaces at the beginning or the end, since leading and trailing spaces are trimmed from all arguments. For example, to print " `and foo =` ", use the command " `echo \ and foo = \` ".

A backslash at the end of *text* can be used, as in C, to continue the command onto subsequent lines. For example,

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

produces the same output as

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

`output` *`expression`*

Print only the value of *expression*, with no newlines or `$`*`nn`*` = `. The value is not entered in the value history.

`output/`*`fmt expression`*

Print the value of *expression* in format *fmt*. Output formats are described in Chapter 5.

`printf` *`string`*`,` *`expressions`* `...`

Print the values of the *expressions* under the control of *string*. *expressions* are separated by commas and may be either numbers or pointers. Their values are printed as specified by *string*, exactly as if the program were to execute

```
printf (string,expressions ... );
```

For example, the following command prints two values in hex:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences you can use in the string are the ones that consist of backslash followed by a letter.

`log`

Write all commands and output to the file iwdb_logfile. At any point during a debugger session, you can enter the `log` command and the debugger will respond with the following message:

```
Command output logging is now ON
```

The debugger will begin writing all commands and output to iwdb_logfile at this point. When you want to stop logging, enter the `log` command again. The debugger will respond with the following message:

```
Command output logging is now OFF
```