About this manual

Audience

This manual is written for those who want to know how to develop and execute programs on the iWarp Computer System. The manual assumes that the reader understands the UNIX operating system and the C programming language.

Using this manual

This manual is both a user's guide and a reference manual, and is intended for both the new and experienced iWarp programmer. The manual provides both tutorial information for new users and reference information for those already experienced with the iWarp system.

If you are a new user, start with Chapter 1 and read through the manual to understand how the iWarp system works and how to use the program development tools to create programs.

If you are familiar with the iWarp system and want to learn about program development, start with Chapter 5, Developing a Program.

For those experienced with programming on the iWarp system, the manual also provides reference information. The manual is designed as follows:

Chapter 1: Provides an overview of the iWarp system.

Chapter 2: Describes the iWarp program development software.

Chapter 3: Describes the iWarp runtime system.

Chapter 4: Describes program communication for the iWarp system.

Chapter 5: Explains how to develop and execute programs on the iWarp system.

Appendix A: Provides a quick-reference of iWarp commands.

Appendix B: Provides descriptions of the PathLib communication routines.

Conventions:

Documentation conventions

- **bold type** is used to reference functions supported by Unix or iWarp man pages.
- *italic type* indicates items you must provide.
- `courier font` is used for program and command examples.

In a syntax description:

- [ ] square brackets enclosing qualifiers indicate the qualifiers are optional.
- { } braces enclosing qualifiers indicate that one or more of the qualifiers is required.
- | a vertical rule separating qualifiers indicates that only one of the qualifiers can be selected.
- ... a horizontal ellipsis indicates that the preceding optional item can appear one or more times in succession.
- ⋮ a vertical ellipsis indicates the continuation of statements within a program example.
- spaces are used between entities to improve readability in both syntax descriptions and programs but unless otherwise noted have no significance.

List of Related Documents

The following documents provide more information about the iWarp system.

- *Introduction to iWarp*
- *iWarp Microprocessor* - A component overview.
- *iWarp C Manual*
- *Apply Manual*
- ***iWarp Macroarchitecture Specification.*

NOTE

A document preceded by ** is proprietary. You must sign a non-disclosure agreement before this document can be released.

1 iWarp System Overview

Building Blocks for High Performance Systems

The heart of the iWarp system is a custom VLSI single-chip processor which consists of approximately 600,000 transistors. This iWarp component serves as a building block for a diverse set of high-performance parallel systems. The iWarp component architecture is designed for high-speed signal, image, and scientific computing and requires only the addition of memory chips to form a complete system building block, the iWarp cell.

iWarp systems can support a wide range of configurations, from special-purpose systolic arrays to general-purpose distributed memory computers. The iWarp system supports both fine-grained and course-grained distributed computation models simultaneously in the same system, a system that can be expanded to include up to 1,024 cells.

The iWarp Component

Before developing programs on the iWarp system, it is helpful to understand the component architecture.

The iWarp component is the basic building block of the iWarp system. Each iWarp component is a single-chip processor that requires only the addition of local memory to form a complete system building block called an iWarp cell. Figure 1-1 illustrates an iWarp cell.

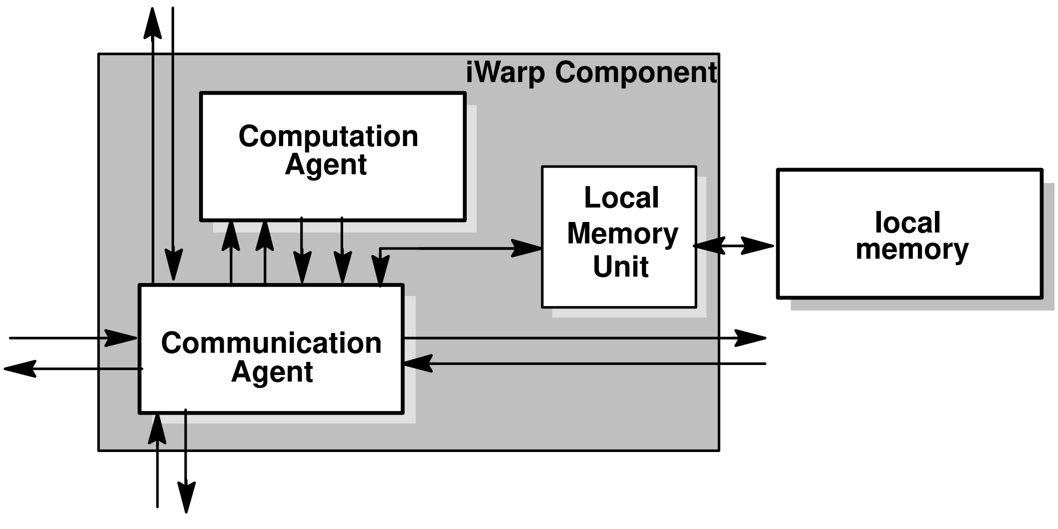


Figure 1-1 : iWarp cell

The iWarp component contains the following three distinct sub-components:

- the computation agent
- the communication agent
- the local memory unit

The computation agent performs the normal, explicitly programmed computation. The communication agent handles input from and output to adjacent cells. The local memory unit provides access to local memory for both the computation and communication agents.

Independent Communication and Computation Agents allow the communication activities of a cell to proceed without disturbing computation. Closely-coupled yet independently-controlled agents make it possible to efficiently overlap communication and computation, and provide greater efficiency for random communication. Nonadjacent cells in the array communicate without disturbing the computation on intermediate cells.

Figure 1-2 provides a block diagram of the iWarp component architecture.

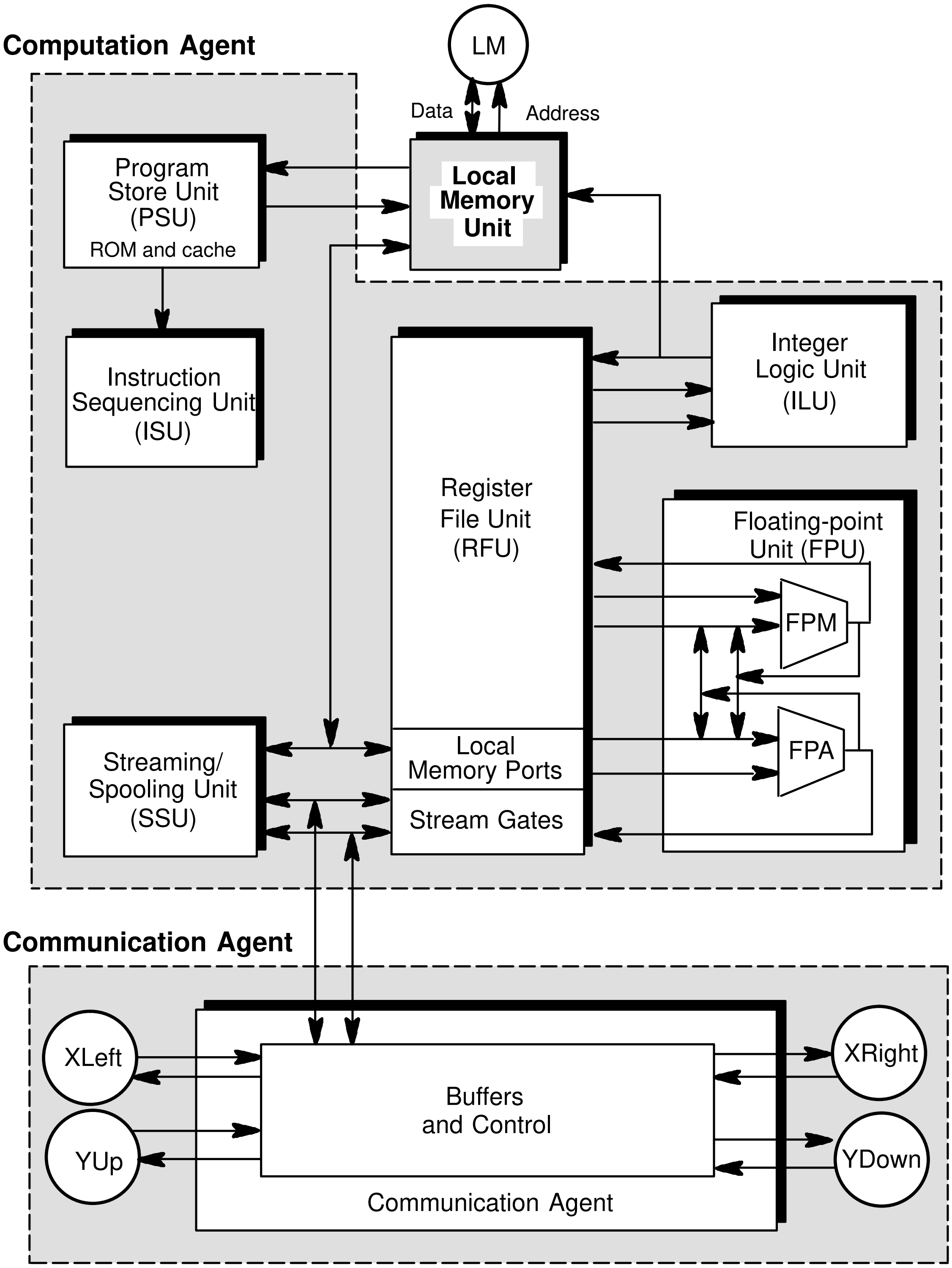


Figure 1-2 : iWarp component architecture block diagram

Communication in the iWarp System

The iWarp system supports both message-based and program-based (systolic) communication, providing support for a wide range of programming models and applications. You are not limited to a single communication style within an application however. When two cells communicate, one can perform message-based communication, while the other performs program-based communication.

The iWarp architecture also has two important innovations to support communication flexibility. These innovations are program access to communication, and logical channels. Chapter 3 provides a detailed discussion of communication in the iWarp system, and describes the communication tools available to the programmer.

NOTE

Message-based communication is not supported in this release of the iWarp software. Message-based communication will be supported in the next software release.

Message-based Communication

In message-based communication (also known as memory-based communication), the unit of communication and processing is a complete message. A complete message is built in the local memory of the sending cell before it is sent to the receiving cell. Only after the entire message has been received in the local memory of the receiving cell does the program running on the receiving cell operate on the data.

In message-based communication, messages are built in the local memories of cells and delivered by independent communication software, so the program running on a cell is not involved in communication. This separation of communication and computation allows a cell to operate on its local data while a message is delivered and buffered through its local memory.

Message-based communication is appropriate for applications that do not assume detailed knowledge about intercell communication. While message-based communication provides the advantages of separate communication and computation and ease of use, these advantages incur higher communication overhead.

Program-based Communication

In program-based communication (also known as systolic communication), the unit of communication and processing is a single word transfer. Each cell processes data immediately as each word arrives. The cell then sends the results of the computation to other cells as data of outgoing messages. This creates a long data stream flowing through the iWarp array.

Program-based communication provides the following advantages:

- Fine-grained communication
- Reduced access to local memory

- Increased instruction-level parallelism
- Reduced size for local memory

With these advantages comes an increased complexity. Message queues in the communication agent of a cell are accessed sequentially, so the programmer must ensure that the reads and writes of message queues are properly sequenced so the correct data item appears at the front of the queue when it is supposed to be read.

The PathLib Communication Library contains routines for implementing program-based communication. Refer to Chapter 4 for a description of PathLib.

Program Access to Communication

For program-based communication, the iWarp system gives programs running on the computation agent direct access to the inputs and outputs of message queues in the communication agent. These locations can be bound to special registers, called gates, in the register file. Reading from a gate corresponds to receiving data from the queue, and writing to a gate corresponds to sending data to the queue.

In addition to supporting program-based communication, direct program access to the message queues can also speed up message-based communication. Message-passing communication is usually implemented by station-to-station delivery , where messages are copied from user space to system space in the sending cell, transmitted, and copied from system space to user space at the receiving cell. Direct access to message queues allows door-to-door delivery, where data is sent directly between the user spaces of the sending and receiving cells.

The iWarp program also provides program access to data routing. Under normal operation, the communication agent establishes a route between the sending and receiving cells, and each word of a message follows the same route. The iWarp system allows a program to alter the route in the middle of a message, so the rest of the data can be sent to another cell using a different route. This program access to data routing reduces the need to buffer data in memory, reducing size requirements for local memory.

Logical Channels

Logical channels allow cells in an iWarp array to be highly interconnected by providing multiple logical connections over a single physical connection. Up to twenty logical channels can be multiplexed over each of the the eight external physical busses in each cell. This allows the implementation of algorithms that require more cell connections than can be physically provided by the hardware. This high degree of connectivity is essential for program-based communication, since each cell may need to have simultaneous connections to several cells.

Logical channels are set up by the program executing on the sending cell. Once a logical channel is established, it remains until it is explicitly taken down by the cell that established it. This long-lived connection decreases communication latency, because once a logical channel is established, no further overhead is associated with the transfer of data. Logical channels allow non-adjacent cells to communicate as if they were adjacent, and they allow efficient simulation of communication patterns that do not map directly onto the underlying hardware.

Logical channels can also be used to guarantee communication bandwidth for special classes of messages between a set of selected cells. The time-multiplexing of logical channels onto physical busses is scheduled such that a minimum bandwidth is guaranteed to be available to each logical channel, and therefore to the messages using that logical channel. Also, when a logical channel is idle, it does not consume any bandwidth, and the bandwidth reserved for it can be used by other logical channels.

Programming the iWarp System

The iWarp system provides a broad base of program development and file manipulation tools to develop application programs, as well as an execution environment for executing programs.

Programming Tools

The iWarp program development tools include the following:

- C compiler
- assembler
- linker
- disassembler
- profiler
- archiver
- symbol table lister
- dumper
- sizer

These tools are described in Chapter 2.

Execution Environment

The iWarp Run Time System (RTS) provides the execution environment for programs on the iWarp system. RTS includes a resident runtime system on each cell and provides a number of services commonly associated with general-purpose microprocessors, such as remote debugging and remote I/O support. RTS also includes the iWarp monitor, which provides an interactive environment to load, execute, and debug programs.

Refer to Chapter 4 for a complete description of RTS.

Parallel Program Generators

Each parallel program is based on a programming model that defines the performance characteristics for the parallel computer. For example, a programming model that can be used for image processing is the domain partition model. With the domain partition model, a problem domain, such as a grid space corresponding to an image, is partitioned so that each cell handles a subdomain.

Typically, when a programmer designs a parallel program, the programmer must build both the program and the programming model. This adds to development time and can result in a program that is difficult to change since the programming model is specific to the program.

For this reason, parallel program generators have been designed to improve parallel programming. Parallel program generators are application-specific tools that translate a user program into programs for each cell in the system, incorporating the low-level constructs required to implement the parallel programming model for that application.

The Apply parallel program generator is available with the iWarp system for image processing applications. Apply allows the programmer to write high-level image processing programs without having to be concerned with the implementation of the programming model.

The Apply Programming language is designed for programming image-to-image computations where the pixels of the output images are computed from corresponding rectangular windows of the input images. Each Apply procedure operates on a single pixel position, and the Apply compiler generates a parallel program that executes the procedure in parallel over the entire image on an iWarp array . An Apply procedure does the following:

- accepts a pixel or window of pixels from each input image
- performs a computation on the input
- returns a pixel value for each output image

The following example illustrates an Apply program that averages pixel values from an input image to get a pixel value for the output image.

```
procedure avg( imagein   : in array (–1..1, –1..1) of byte border 0,
               imageout  : out byte)
is
begin
    imageout := (imagein(–1,1) + imagein(0,0) + imagein(1, –1)) / 3;
end avg;
```

Figure 1-3 provides an illustration of the operation performed by the Apply program in the example.

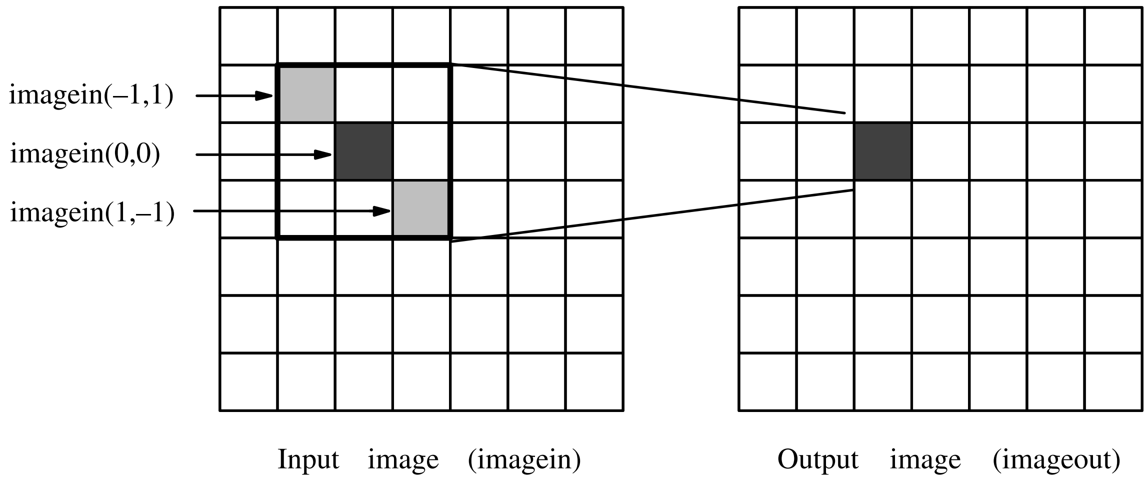


Figure 1-3: Apply Averaging Operation

You call Apply procedures from a C main program, passing the input and output images and other parameters as arguments. On the iWarp system you can generate both single-cell (sequential) programs and multi-cell (parallel) programs with Apply.

For a complete description of the Apply language and parallel program generator, refer to *The Apply Manual.*

The following chapter, *iWarp Program Development Tools*, provides descriptions of the tools used to develop programs on the iWarp system.

2 The Program Development Environment

Introduction

The iWarp programming tools reside on the iWarp host, which is a Sun Microsystems ™ UNIX-based workstation. The program development environment includes the following tools for developing programs:

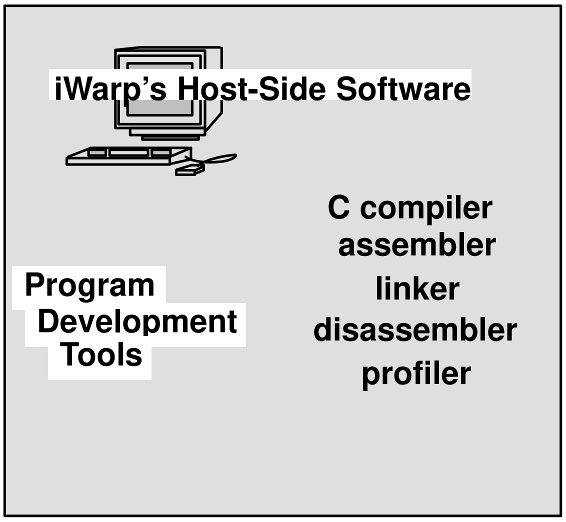
- C compiler
- assembler
- linker
- disassembler
- profiler

The program development environment also includes the following tools for file manipulation:

- archiver
- lister
- dumper
- sizer

The following sections describe these tools and the commands and options used to invoke them.

Program Development Tools



iWarp C

iWarp C is a superset of the C language as defined in the 1978 edition of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie. iWarp C contains several iWarp-specific extensions, as well as a number of widely recognized extensions. The following list summarizes the iWarp-specific extensions:

- Communication support (send and receive primitives).
- Condition code checking support (iWarp condition codes).
- Assembly code inlining facility.
- Pragma support for inlining specification.

Refer to the *iWarp C Manual* for a complete description of the iWarp-specific implementations of the C language and the iWarp C extensions. For a complete description of the C programming language, refer to *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, 1978, Prentice-Hall.

The iWarp C compiler is an optimizing compiler for the iWarp processor. The compiler is a cross-compiler, running on the Sun host and producing code for the iWarp system. The iWarp C compiler translates C programs into iWarp executable load modules or iWarp relocatable binary programs for loading with the iWarp linker . The iWarp C compiler produces object code files in Common Object File Format (COFF).

Invoking the C Compiler: iwcc

To invoke the iWarp C compiler, use the iwcc command as follows:

```
iwcc [option ...]  filename ...
```

filename can be any of the following:

- A C source file (.c file extension).
- A C source file that is not to be run through the preprocessor (.i file extension).
- An assembly language source file (.s file extension). These files are passed to the assembler and an object file is produced.
- An intra-file inlined intermediate file (.K file extension). These files are used by the global optimizer for inter-file inlining of any specified C programs.

All other files are passed to the linker.

option can be any of the following options.

iwcc Options

-#	List each phase of compilation being executed and the command options used to invoke the phase.
-##	Same as -#, but also prints the full path used to load each phase of the compilation.
-###	Same as -##, but do not execute any phases of the compilation.
-c	Suppress the link edit phase of the compilation and produce an object file for each source file.

-C	Prevent the C preprocessor from removing comments.
-D*name* [=def]	Define *name* for the preprocessor as if by a #define directive. If =def is not included, *name* is defined as 1. If both the -D and -U options are used with the same *name* , *name* will be undefined.
-E	Run only the preprocessor on the named C programs and send the result to standard output. The output includes cpp-type line number information.
-fasmwiden	Convert floating-point operands to double-precision operands for asm insert function parameters and function return types.
-fnoasmwiden	Suppress the conversion of floating-point operands to double-precision operands for asm insert function parameters and function return types. This is the default.
-fnowiden	Suppress the automatic conversion of floating-point operands to double-precision operands within expressions, and suppress the conversion of floating-point parameters to functions and function return types. This option is not recommended, and there is no library support for float parameters or function returns.
-fsingle	Suppress the automatic conversion of floating-point operands to double-precision operands within expressions.
-g	Generate information for symbolic debugging. This suppresses all optimization.
-G*c*	Retain intermediate files from pass *c* for debugging, where *c* is one of the following: p preprocessor 0 compiler first pass q Portable Code Optimizer (PCO) pass 1 compiler second pass a assembler l link editor m listing merger
-I*pathname*	Add *pathname* to the list of directories in which to search for #include files with relative pathnames (not beginning with /). The preprocessor first searches for #include files in the directory containing the source file, then in directories named with the -I option, then in /iwarp/include.
-K	Invoke PCO to generate an intra-file inlined intermediate file. Arguments whose names end with .c are compiled, intra-file inlined by PCO, and each intermediate file is left in a file whose name is that of the source file with .K substituted for .c. The output file name cannot be changed with the -o option.
-l*x*	This option is an abbreviation for the library name lib*x*.a, where *x* is a string. This option is used with the -L option.

-L*dir*	Add *dir* to the list of directories containing object library routines. Directories specified with -L are searched before the standard directory, /iwarp/lib. This option only affects -l*x* options which follow on the command line.
-M	Merge the C and generated iWarp assembler files into a file suffixed .m. For a single file, the output file name can be changed with the -o option.
-MM	Same as -M, but includes the text of #include files in the listing file.
-N*letter*	*letter* can be C, L, or S. C suppresses the automatic use of a default link control file. L suppresses automatic linking with libc.a (and libg.a if -g is used). S suppresses the automatic linking with iwcrt0.o (or iwmcrt0.o if -p is used).
-o *filename*	Specifies *filename* as the name of the object file, overriding the default name a.out. Use with -c modifies the name of the object file. Use with -P modifies the name of the preprocessed file. Use with -S modifies the name of the output assembly file. Use with -M modifies the name of the output merged listing file.

-O[*level*]

Invoke optimization as indicated by *level*, where *level* is one of the following:

0	no optimizations
1	global optimizations on intermediate code (PCO)
2	basic block compaction generation of Compute and Access instructions, as well as PCI and PCO optimizations
3	software pipelining, basic block compaction, and PCO optimizations

NOTE

Software pipelining and basic block compaction will be available in the next release of the software.

If *level* is not specified, this option will invoke all available optimizations.

-p	Produce code that counts the number of times each routine is called. If link editing takes place, replace the standard startoff routine by one that automatically calls the monitor at the start and arranges to write a mon.out# file at normal termination of execution of the object program (# is the cell number). This allows generation of an execution profile with the iwprof command.
-P	Run only the preprocessor on the specified C programs and leave results in the corresponding files suffixed .i. For a single file, the output file name can be changed with the -o option.

-Q*options*	Invoke optimizations as defined by *options* , where *options* can be any or all of the following: G invoke global optimizations (PCO) C compaction of basic block for Compute and Access instruction S software pipelining **NOTE** Software pipelining and basic block compaction will be available in the next release of the software. If compile time is an issue, but you want to do some optimization, use the -O*level* option to reduce the amount of optimization done.
-S	Compile the specified C programs and leave the assembler-language output in corresponding files suffixed .s. For a single file, the output file name can be changed with the -o option.
-u*name*	Causes the linker to enter *name* as an undefined symbol in the symbol table.
-U*name*	Remove any initial definition of *name* , where *name* is a reserved symbol predefined by the preprocessor.
-V	Display the version of the C compiler and all invoked passes, and perform compilation as specified.
-VV	Display the version of the C compiler and all possible passes, but do not perform compilation.
-Wc, *arg1* [, *arg2* ...]	Hand off any specified arguments to pass *c*, where *c* is one of the following: p preprocessor 0 compiler first pass q PCO pass 1 compiler second pass a assembler l link editor m listing merger
-x	Preserve only global (non-.globl) symbols in the output symbol table. Enter only external symbols.
-Yc, *dir*	Get *c* from directory *dir* , where *c* is one of the following: p preprocessor 0 compiler first pass q PCO pass 1 compiler second pass

a	assembler
l	link editor
m	listing merger
C	link.ctl
L	default search directory for libraries (/iwarp/lib)
S	startup code (iwcrt0 or iwmcrt0)

The iWarp assembler

The iWarp assembler is a cross-assembler, running on the Sun host and producing code for the iWarp system. You can use the assembler directly to translate assembly language programs, or indirectly as a compiler option. The assembler output file contains the following:

- binary iWarp operation codes
- relocation information
- a symbol table
- symbolic debugging information

Assembler output is suitable for processing by the iWarp linker. The assembler produces object code files in COFF.

Invoking the Assembler: iwas

To invoke the iWarp assembler, use the iwas command as follows:

`iwas [`*`option`*` ...]  `*`filename`*

filename is the name of an assembly language source file. Only one source file may be specified.

option can be any of the following options.

Assembler Options

-#	Print the active command line options.
-##	Print the active command line options and a message listing permitted command line options.
-dl	Remove the line number entries from the symbol table of the object file.
-L	Create a listing file with the extension .l.
-m	Use the m4 preprocessor on the source file. Uses macro definitions in /iwarp/lib/cm4defs by default.
-o *filename*	Specifies *filename* as the name of the object file, overriding the default name.
-O	List intermediate file information for debugging.
-rom	Prohibit data from straddling a 64-byte boundary. The default does not allow straddling a 256-byte boundary.
-unlink	Remove the source file on completion of assembly. Useful when assembly is used in a chain of programs, as with **iwcc** .

-V	Write the version number of the assembler
-VV	Write the version number of the assembler on standard error output and suppress assembling the source file.
-W	Suppress warning messages.
-XA	Enable the use of the alternate register names r0-r127 (general purpose registers) and cs000-cs3ff.
-XD	Dump the internal tables of machine-specific names. The file named on the command line is not assembled.
-XN	Suppress printing of the date on listings.

The iWarp linker

The iWarp linker combines object files into a single file. The iWarp linker is a cross-linker that executes on the SUN host. This linker maps input files onto output files and leaves a completely linked object file for the iWarp loader. Key features of are the linker are:

- links multiple object files into one single object program
- resolves external global addresses
- provides a special command language to specify explicit link instructions
- provides a default memory map that matches the logical address space
- allows the user to override the default memory layout and to specify a different layout of memory, into which the linked object is to be mapped
- binds output sections to default machine addresses
- binds output sections to specific addresses, overriding the defaults
- places sections relative to one another through explicit commands
- binds individual names to specific addresses per explicit commands
- creates a load map
- associates input sections with specific addresses chosen by the user

Invoking the Linker: iwld

To invoke the iWarp linker, use the iwld command as follows:

```
iwld [option ...]  filename ...
```

filename is the name of an object file.

option can be any of the following options.

Linker Options

-e *entry*	Define *entry* as the entry point of the program.
-l*x*	This option is an abbreviation for the library name lib*x*.a, where *x* is a string. This option is used with the -L option.
-L*dir*	Add *dir* to the list of directories containing object library routines. Directories specified with -L are searched before the standard directory , /iwarp/lib. This option only affects -l*x* options which follow on the command line.

-m	Produce a simple load map, listing the names of the files which will be loaded plus their start addresses, input sections, output sections, and lengths.
-o *filename*	Specifies *filename* as the name of the object file, overriding the default name a.out.
-r	Prevent final definitions from being given to common symbols, and suppress undefined symbol diagnostics.
-s	Removes the symbol table and relocation information to save space.
-u *name*	Enter *name* as an undefined symbol.
-x	Preserve only global symbols in the output symbol table.

The iWarp disassembler

The iWarp disassembler translates a machine language file and produces an assembly language source file suitable for the iWarp assembler. The input must be a COFF file.

Invoking the Disassembler: diwas

To invoke the iWarp disassembler, use the diwas command as follows:

```
diwas [option ...]  filename
```

filename is the name of a COFF object file. Only one file may be specified.

option can be any of the following options.

Disassembler Options

-#	Print version information and executes disassembly.
-##	Print version information, but does not execute disassembly.
-b	Produce byte mode addresses. This is the default.
-B	Suppress printing of byte mode addresses.
-c	Decode the given disassembly range as instructions rather than data.
-d	Dump the symbol table for debugging. A list of the symbols contained in the symbol table is included in the output. Comment markers precede the list of symbol table entries so that the source code produced by the -q option can still be directly assembled.
-D	Produce listing style output using obsolete double word addresses.
-f	Treat the input file as if the object file being disassembled had been assembled with the .flags psuedo-operation. When the -f option is used, all instructions which do not affect flags are preceded by the prefix *noflags* , while instructions which do affect flags have no prefix. The default behavior is the reverse: instructions which do not affect status flags have no prefix, while instructions which do affect flags have the prefix *flags* .
-H *path*	Define *path* as the path to the directory where the disassembler creates its symbol and hash table.

-i	Force the specified disassembly range to be initialized data rather than separate data and text sections.
-l	Invoke listing mode, producing hexadecimal location and object code output. This is the default.
-L *path*	Define *path* as the path to the standard symbol mnemonics (registers, condition codes).
-n	Create a new hash table for symbols.
-o *filename*	Specify *filename* as the name of the disassembly output file, overriding the default name.
-q	Suppress listing mode, causing the output to look like assembly language source code.
-r	Disable symbolization of addresses. The disassembler prints only addresses, not labels, as the operands of instructions.
-R *start:end*	Define a range between address *start* and address *end* for disassembly. *start* and *end* are hexadecimal byte addresses. The default for *start* is 0, and the default for *end* is infinity. The -R option suppresses headers and trailers.
-s	Enable symbolization of addresses. The disassembler uses what symbol table information is available to try to replace numeric addresses by labels in instructions. This is the default.
-t	Print elapsed user and system times for the phases of disassembly: load, startup, symbol table initialization, and code disassembly.
-u	Reuse the symbol hash table constructed at the last invocation of the disassembler .
-w	Produce listing style output using obsolete word addresses.
-W	Specify that the labels stored in the symbol table are given as word addresses.
-XA	Handle unknown condition codes and control space registers by using generic names.
-XD	Compile the standard symbol names from the text output produced by the assembler into a fast binary format.

The iWarp profiler

The profiler generates profile information from a file produced by the monitor function. The profiler lists the percentage of time spent executing between the address of each external symbol and the following symbol. The profiler also lists the number of times each function was called and the average time per call.

You must use the iwcc -p option when compiling to generate a profile file.

Invoking the Profiler: iwprof

To invoke the iWarp profiler, use the iwprof command as follows:

```
iwprof [option ...] filename
```

filename is the name of an object file.

option can be any of the following options.

Profiler Options

-a	Sort by increasing symbol address.
-c	Sort by decreasing number of calls.
-h	Suppress the heading normally printed on the report.
-m *mdata*	Use the file *mdata* instead of mon0.out as the input profile file.
-n	Sort lexically by symbol name.
-o	Print each symbol address (in octal) along with the symbol.
-s	Print a summary of the monitoring parameters and statistics on the standard error output.
-t	Sort by decreasing percentage of total time. This is the default.
-V	Print iwprof version information on the standard error output.
-x	Print each symbol address (in hexadecimal) along with the symbol name.
-z	Include all symbols in the profile range, even if associated with zero number of calls and zero time.

File Manipulation Tools

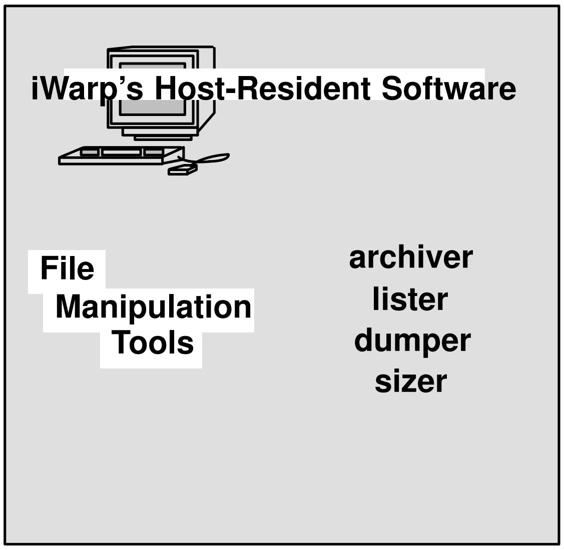
The iWarp file manipulation tools operate on COFF files. COFF is an acronym for Common Object File Format. The COFF definition describes a complex data structure that represents object files, executable files, and archive files. The COFF data structure defines fields for things such as machine code, relocation information, and symbolic information.

Key features of COFF are:

- applications may add system-dependent information to the object file without causing access utilities to become obsolete
- COFF provides space for symbolic information used by debuggers and other applications.
- users may make some modifications in the object file construction at compile time

If you are unfamiliar with UNIX Common Object File Format and want to know more, we suggest two books;

- *Understanding and Using Coff* by Gintaras R. Gircys, O'Reilly and Associates, Inc..
- UNIX System V/386 Programmer's Guide, Prentice Hall.



The iWarp Archiver

The iWarp archiver maintains groups of files combined in a single archive file. Its main purpose is to create and update library files to be used by the linker. The magic string and the file headers used by the archiver consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

Invoking the Archiver: iwar

To invoke the iWarp archiver, use the iwar command as follows:

`iwar key [posname] [option ...] afile [filename ...]`

posname is the name of a file in the archive used as a reference point to position other files in the archive.

afile is the name of an archive file.

filename is the name of a file within the archive file.

A single *key* must be included, and it can be any of the following:

Archiver Keys

d	Delete the named files from the archive file.
m	Move the named files either to the end of the archive, or the position specified by a positioning argument and *posname* as in the case of the r key.
p	Print the named files. If no names are given, all files in the archive are written to the standard output in the order they appear in the archive.
q	Append the named files to the end of the archive file without searching the archive for duplicate names. This key is useful only to avoid quadratic behavior when creating a large archive piece-by-piece. If this key is used to add a member to an archive, and a member with the same name as that member already exists in the archive, the old member will not be removed; two members with the same name will exist in the archive. Optional positioning characters are invalid with this key.

r	Replace the named files in the archive. If the optional character u is used with r, then only those files with dates of modification later than the archive file are replaced. If an optional positioning character a, b, or i is used, *posname* must be present to specify that new files are to be placed after or before *posname* . Otherwise, new files are placed at the end.
t	Print a table of contents of the archive file. If no names are given, all files in the archive are listed.
x	Extract the named files. If no names are given, all files in the archive are extracted into the current directory. The archive is not altered.

option can be any of the following options.

Archiver Options

-a	Place new files after *posname* . This applies only to the r and m keys.
-b	Place new files before *posname* . This applies only to the r and m keys.
-c	Create an archive. Suppress the message that is produced by default when the archive is created.
-i	Identical to the b option.
-l	Place temporary files in the current working directory rather than in the default temporary directory, TMPDIR.
-s	Force the regeneration of the archive symbol table even if the archiver is not invoked with a command that will modify the archive contents. This command is useful to restore the archive symbol table after the strip command has been used on the archive.
-u	Replace those files that have changed since they were put in the archive. -u is used with the r key.
-v	Use verbose mode. When used with the r, d, m, or q keys, give a file-by-file description of the creation of a new archive file form the old archive and the constituent files. When used with x, give a file-by-file description of the extraction of archive files. When used with t, give a long listing of information about the files. When used with p, write each member's name to the standard output before writing the member to the standard output.

The iWarp symbol table lister

The symbol table lister displays the symbol table of common object files. The object files can be relocatable or absolute common object files. They can also be archives of relocatable or absolute common object files. For each symbol, the following information is printed;

- The name of the symbol.
- The value of the symbol, expressed as an offset or an address depending on its storage class.
- The storage class of the symbol.
- The type and derived type of the symbol.

- The size of the symbol in bytes (if compiled with the iwcc -g option).
- The source line number at which the symbol is defined (if compiled with the iwcc -g option).
- The object file section that contains the symbol (for storage classes static and external).

Each symbol name in the list is preceded by its value and one of the following:

A	absolute
B	bss segment
C	common symbol
D	data segment symbol
f	filename
t	static function symbol
T	text segment symbol
U	undefined

Invoking the Symbol Table Lister: iwnm

To invoke the iWarp symbol table lister, use the iwnm command as follows:

`iwnm [`*`option ...`*`]` *`filename ...`*

filename is the name of an object file or an archive of object files.

option can be any of the following options.

Symbol Table Lister Options

-e	Print only external and static symbols.
-f	Produce full output. Print redundant symbols (.text, .data, .lib, and .bss) normally suppressed.
-h	Do not display the output header data.
-n	Sort external symbols by name before they are printed.
-o	Print the value and size of a symbol in octal instead of decimal.
-p	Produce easily parsable, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters U (undefined), A (absolute), T (text segment symbol), D (data segment symbol), S (user defined segment symbol), R (register symbol), F(file symbol), or C (common symbol). If the symbol is local (non-external), the type letter is in lower case.
-r	Prepend the name of the object file or archive to each output line.
-T	Truncate every name which would otherwise overflow its column and place an asterisk as the last character in the display name to mark it as truncated. By default, iwnm prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned.

-u	Print only undefined symbols.
-v	Sort external symbols by value before they are printed.
-V	Print the version of the iwnm command executing on the standard error output.
-x	Print the value and size of a symbol in hexadecimal instead of decimal.

The iWarp dumper

The dumper dumps selected parts of object files. The dumper processes both object files and archives of object files.

Invoking the Dumper: iwdump

To invoke the iWarp dumper, use the iwdump command as follows:

```
iwdump [option ...] filename ...
```

filename is the name of an object file or an archive of object files.

option can be any of the following options.

Dumper Options

-a	Dump the archive header of each member of each archive file specified.
-c	Dump the string table.
-f	Dump the header file.
-g	Dump the global symbols in the symbol table of an archive.
-h	Dump section headers.
-l	Dump line number information.
-L	Interpret and print the contents of the .lib sections.
-o	Dump each optional header.
-r	Dump relocation information.
-s	Dump sections contents.
-t	Dump symbol table entries.
-z *name*	Dump line number entries for the named function.

The following modifiers can be used with the iwdump options:

Dumper Option Modifiers

-d *number*	Dump the section number, number, or the range of sections starting at number and ending the number specified by +d.
+d *number*	Dump sections in the range either beginning with first section or beginning with section specified by -d.
-n *name*	Dump information pertaining only to the named entity. This modifier can be used with -h, -s, -r, -l, -t

-p	Suppress printing of the headers.
-t *index*	Dump only the indexed symbol table entry. The -t used in conjunction with +t, specifies a range of symbol table entries.
+t *index*	Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the -t modifier.
-u	Underline the name of the file for emphasis
-v	Dump information in symbolic representation rather than numeric. This modifier can be used with all the above options except -s and -o options of iwdump.
-z *name, number*	Dump line number entry or range of line numbers at *number* for the function *name* .
-+z *number*	Dump line numbers starting at either function *name* or *number* specified by -z, up to *number* specified by +z.

The iWarp sizer

The sizer produces section size information in bytes for each loaded section in common object files. The size of the text, data, and bss sections are printed, as well as the sum of the sizes of these sections. The sizer will also produce size information for all members of an archive. The size of uninitialized non-static global data structures is not included in the size information.

Invoking the Sizer: iwsize

To invoke the iWarp sizer, use the iwsize command as follows:

```
iwsize [option ...]  filename ...
```

filename is the name of an object file or an archive of object files.

option can be any of the following options.

Sizer Options

-f	Print the size of every loaded section, followed by the section name in parentheses.
-n	Include NOLOAD sections in the size.
-o	Print numbers in octal.
-V	Supply version information on the iwsize command.
-x	Print numbers in hexadecimal.

3 The iWarp Runtime System

The iWarp system provides the program developer with a resident runtime system kernel to manage resources and provide services that give the user more control at program execution time. The iWarp Runtime System (RTS) provides the following utilities for loading, executing, and debugging programs:

rtslink	links RTS startup code to a compiled user program
lgo	loads and executes a program
iwmon	provides an interactive environment for loading, executing, and debugging

Each of these utilities is described in a later section.

RTS also supports limited versions of the following system calls on the SIB and array cells:

ioctl()	supports isatty() only
close()	supports cleanup code in libc
exit()	supports the iwmon wait command, the -w option to lgo, and notifies the host monitor when a cell exits
read()	reads from fd 0 to a buffer
write()	writes a buffer to fd 1 or fd 2

There is no support for open(). Applications have stdin, stdout, and stderr open by default.

RTS also supports malloc(), free(), and realloc(), but supplies them as system calls rather than Section 3 library functions.

Memory Allocation

The memory map for an iWarp cell is as follows:

0x0..0x1fff	on-chip ROM
0x2000..0x120ff	RTS
0x12100..end	user program, followed by heap and stack

The memory map for the SIB is as follows:

0x0..0x1fff	on-chip ROM
0x2000..0x7fff	diagnostic array kernel EPROM
0x8000..0x200ff	RTS
0x20100..end	user program, followed by heap and stack

The Stack and the Heap

The system stack is initialized by RTS to the top of memory, 0x7fffc. The heap is initialized to a round value following the user program's bss section. The stack low limit (SPLO) is always the top of the heap. As malloc() needs space, it calls a combination of internal versions of sbrk() and brk(). Brk() adjusts SPLO up, but within reason, that is, never let SPHI – SPLO be less than 8K bytes, don't move SPLO above SP, and so on. If memory is exhausted, a stack low-limit event occurs, indicating that the stack and heap collided.

Once the top of the heap is incremented, that memory remains the property of the heap. SPLO is not decremented when free() is used to deallocate memory. This means that the allocation of a large portion of the heap, however briefly it may be needed, limits the size of the stack for the duration of the application.

Preparing an Executable File: rtslink

To create an executable load module for the iWarp system, you must combine RTS startup code with your compiled program. rtslink prepares a loadable COFF file from a compiled user program by adding the RTS startup code. The rtslink command has the following syntax:

rtslink [*option* ...] *filename* [*arglist*]

filename is the name of the compiled user program.

arglist is any command line arguments to be passed to the user program.

option can be any of the following:

-o *filename*	Defines the name of the loadable COFF file. The default name is rts.*pid,* where *pid* is the process id.
-sib	Specifies that the program is to run on the interface cell. The default is to run on an array cell.
-V	Prints version information and continues.
-VV	Prints version information and exits.

In the following example, rtslink creates the loadable file a.run from the compiled user file a.out:

```
rtslink –o a.run  a.out
```

Loading and Executing Programs: lgo

lgo loads and runs a program on the iWarp system by doing the following:

- locking the specified array
- creating a script that specifies a sequence of load and run commands
- invoking the iWarp monitor and passing the script to the monitor
- unlocking the array when execution has completed

NOTE

The modules being loaded and executed with lgo must first be linked to the RTS startup code by rtslink.

The lgo command has the following syntax:

```
lgo [option ...] device
```

device specifies an iWarp interface cell to which the array is attached.

option can be any of the following:

-f *filename*	Executes iwmon commands from the file specified by *filename* . This option cannot be used with the -r or -w options.
-n	This option is used with -o to generate an iwmon script file without executing the program.
-o *filename*	Outputs the iwmon script to the file specified by *filename* .
-R *file id*	Generates commands in the iwmon script to load and run *file* on the interface cell specified by *id*.
-r *file id_list*	Generates commands in the iwmon script to load and run *file* on the cells specified by *id_list* , where *id_list* has the following syntax: `id [ .. id ]` where *id* is a cell id number.
-w *n*	Waits for *n* cells to exit before returning control to iwmon.
-s	Forces stdin, stdout, and stderr to be redirected from or to the files stdin*c*, stdout*c*, and stderr*c*, where *c* is the cell number of the cell making the I/O request.
-V	Prints lgo version information and continues.
-VV	Prints lgo version information and exits.

In the following example, lgo loads and runs hello.out on cell 0 of the array attached to /dev/isib0:

```
lgo -r hello.out 0 -w 1 /dev/isib0
```

In the following example, lgo loads and runs hello.out on all cells of a 64-cell array attached to /dev/isib1:

```
lgo -r hello.out 0..63 -w 64 /dev/isib1
```

In the following example, lgo generates the script lgo.script for running hello.out on cells 0 and 10:

```
lgo -n -r hello.out 0 10 -o lgo.script -w 2 /dev/isib0
```

In this example, lgo runs the script generated in the previous example on the array attached to /dev/isib1:

```
lgo -f lgo.script /dev/isib1
```

The iWarp Monitor: iwmon

The iWarp monitor provides an interactive environment to load, execute, and debug programs. The monitor allows you to deposit and examine values in cell local memory, general-purpose registers, and control space registers. It also provides for the use of breakpoints.

Invoking the Monitor

To invoke the iWarp monitor, use the iwmon command as follows:

```
iwmon [option ...] [sibnum]
```

sibnum is the number of the Sun Interface Board (SIB) to which the iWarp array is attached.

The SIB number is set when the iWarp system is installed. The SIB number will default to zero if *sibnum* is not specified on the command line. If no SIB exists at zero, iwmon will issue an error message, and you will have to specify an SIB number.

More than one iWarp system can be attached to the same SUN workstation. If this is the case each SIB will have a unique SIB number and the monitor can be used to interact with any of the iWarp systems.

option can be any of the following:

-b *bootfile*	Boots the array-resident portion of iwmon from *bootfile*, instead of from the default package.
-l *logfile*	Specifies *logfile* as the file to receive logging output generated by the programs on the array.
-s	Instructs the monitor to use stdfiles for standard output and standard error. Output on the stdout and stderr streams from cell *n* is directed to files stdout*n* and stderr*n* in the current working directory. Input to the cells stdin streams is taken from stdin*n*.

The following interactive commands are available when you invoke the monitor:

Monitor commands

?	Lists all available commands offered by *iwmon.*
boot [*filename*]	Boot, download, and run the array-resident monitor. If *filename* is not included, the default bootfile is used.
ex *cellid addr* [*count*]	Examine cell memory. Displays *count* number of words of memory on cell *cellid* from memory location at word address *addr*. If *count* is not supplied, the monitor examines one word.
exr *cellid rfaddr*	Examine register file. Examines the word at *rfaddr* in the register file on cell *cellid.*
exc *cellid csaddr*	Examine control space register. Examines the word at *csaddr* in the control space on cell *cellid*.

dep *cellid addr data* [*count*]	Deposit into cell memory. Deposits *data* on cell *cellid* in *count* memory locations starting at address *addr* . If count is not supplied, *data* is deposited in one word.
depr *cellid rfaddr data*	Deposit into register file. Deposits *data* into the word at *rfaddr* in the register file on cell *cellid.*
depc *cellid csaddr data*	Deposit into control space register. Deposits *data* into the word at *csaddr* in the control space on cell *cellid.*
load *cellid filename*	Load a program into cell memory. Loads the COFF file *filename* on cell *cellid* at the location determined by the COFF information in the file. The file must be linked using rtslink.
run *cellid addr*	Run a program. Starts execution of a program on cell *cellid*. The execution starts at address *addr* .
sbp *cellid addr*	Set a breakpoint. Sets a breakpoint at address *addr* on cell *cellid.* A status line is printed that indicates the breakpoint number, which can later be used to unset the breakpoint. Up to 12 breakpoints can be set on each cell. Breakpoints will not work if set at a Compute and Access instruction or any flow of control instruction (including call, return, and end loop).
ubp *cellid bp*	Unset a breakpoint. Unsets breakpoint *bp* on cell *cellid.*
status *cellid*	List breakpoints. Lists currently set breakpoints and their corresponding addresses.
cbp *cellid bp*	Continue execution. Continues execution from breakpoint *bp* on cell *cellid* .
wait *N*	Suspend processing. Suspends command processing until *N* exit messages are received from the array.
log [[-a] *filename*]	Redirect messages. Redirects monitor-based log messages to a file. The -a option appends messages to the file, otherwise, a new file is created. By default, log messages are sent to stdout.
exit	Exit the iWarp monitor.
quit	Same as exit.

Input Conventions

The monitor commands are subject to the following conventions:

- The monitor accepts numeric arguments in decimal, hexadecimal, or octal bases.
- Local memory locations must be specified as word-aligned, byte addresses. Each cell is assumed to have 512K bytes of memory available. The monitor accepts values in the range of 0 to 0x7FFFC.

NOTE

The monitor can handle memory configurations of up to 1.5 Mbytes.

- Register file addresses must be word-aligned, byte addresses. The monitor accepts values in the range of 0 to 0x1FC. The monitor also recognizes register file names defined by the iWarp assembler.
- Control space addresses must be byte addresses. The monitor accepts values in the range 0 to 0x1ff. The monitor also recognizes control space register names defined by the iWarp assembler.
- Cell id numbers range from 0 to the number of cells in the array, including the SIB. The SIB cell id is always equal to the number of cells in the array
- The maximum count for examine and deposit memory commands is 128.
- File names are SunOS pathnames.

Monitor Example

The following is an example of running a single-cell program on the iWarp system using the iWarp monitor. Explanations of the commands used in the example follow the example.

```
sunhost> siblock 1

sunhost> iwmon 1
```

iWarp Monitor V2.1 Mon July 1 14:20:24 PST 1991 coff V2.1

```
resetting the array...

booting /iwarp/lib/iwmon.package...

iwmon>load 2 myprog

iwmon>run 2 4000

iwmon>exit

sunhost> sibunlock 1
```

The commands shown in the example perform the following functions:

siblock	locks the iWarp SIB interface for exclusive use.
iwmon	starts the monitor program and initializes the iWarp array.
load 2 myprog	loads the user program “myprog” on cell-id “2” of the iWarp array .
run 2 4000	runs the program loaded on cell-id “2” starting at word-address “4000” hex.
quit	exits the iWarp monitor.

sibunlock unlocks the iWarp SIB interface and frees it for use by others.

4 Program Communication: PathLib

Chapter 1 provided a brief overview of program communication on the iWarp system. This chapter describes the PathLib library of communication functions used to implement program-based communication. Appendix B provides thorough descriptions of all PathLib functions. The section *PathLib Terms*, at the end of this chapter, provides definitions for PathLib terminology.

NOTE

> It is essential to have a clear understanding of the iWarp architecture to effectively design programs using the PathLib communication functions. For a complete description of the iWarp architecture, refer to the *iWarp Macroarchitecture Specification*.

Communication Overview

Each cell's communication agent maintains a hardware structure called the Pathway Control Table (PCT). Together, the communication agent and PCT provide the mechanisms necessary to support logical channels, the means by which cells communicate over the array.

The PCT contains a collection of PCT entries. The PCT entry is the fundamental resource comprising a logical channel. A logical channel is a chain of PCT entries that link a sender cell to a receiver cell. All cells in the path across the array from the sender cell to the receiver cell contribute PCT entries to support a logical channel, as do the sender and receiver cells.

As a logical channel is being created, it competes with other logical channels being created for the resources on the cells it passes through to reach its destination. Once resources are allocated to a logical channel, they carry data sent over this logical channel until the logical channel is removed by the deallocation of its PCT resources.

Once created, a logical channel receives a guaranteed share of the capacity of the pathways over which it flows. How much of the capacity any one channel receives depends on the number of channels competing for capacity on the same physical channel. No capacity is required if no data is moving over a logical channel.

The communication agent PCT contains twenty entries, providing hardware support for a maximum of twenty simultaneous logical channels per cell. These resources are shared between the runtime system and PathLib, so less than twenty are available to the PathLib user. The exact number depends on the runtime system services operating on the cells.

For more information about the iWarp processor architecture, refer to the *iWarp Macroarchitecture Specification*.

PathLib Overview

PathLib supports the following communication operations:

- The creation and termination of logical channels, called connections.
- User creation and termination of messages over logical channels, known as messages and marks.
- User management of streamgates.
- Spooled block data movement over logical channels, called spooling.
- The creation and termination of aggregate logical channels, called joining and splitting, and known in PathLib as express.

Connections

A connection is an instance of a logical channel created by PathLib. A connection is known only at its endpoints: one on the origin cell of the connection and one on the destination cell. Connections are created and destroyed via the production and consumption of connection headers and trailers. The PathLib connections functions support the creation and termination of connections.

PathLib includes the following connection functions:

pl_create_connection()	Creates a connection with the calling cell as its origin and the route and destination specified via the user-supplied arguments. The connection is created by the sending of a connection header encoded with routing and destination information.
pl_accept_connection()	Accepts a connection header on the calling cell. The use of this assumes that some cell sent a connection with this cell as its destination. Upon return from this function, the creation of the connection is complete and all resources to support it are established.
pl_destroy_connection()	Destroys a connection originating from the calling cell. The communication agent resources that comprise the connection are deallocated by the sending of a connection trailer.
pl_conclude_connection()	Concludes a connection on the calling cell. The connection trailer is consumed, thus freeing the last resources associated with the connection.

The index of the PCT entry for a connection is returned when a create or accept connection function is called. This value is the users handle on the connection. The user is expected to supply this value to identify connections to PathLib functions that perform operations on connections.

Messages

A message is a block of data transmitted over a connection. All data must be sent within the context of a message. It is up to the user to establish messages by sending and receiving message headers, and to end messages by sending and receiving message trailers.

PathLib includes the following message functions:

pl_produce_message_header()	Starts a message by sending a message header. The message header will follow the path of the connection it is sent upon until it is stopped by a cell along the connection, or until it reaches the ultimate destination of the connection or aggregate logical channel of which the connection is a constituent (refer to the description of express in Appendix B). A message header may be stopped or consumed by any cell along the connection upon which it travels. Cells along the connection may request that generic message headers be stopped, or a message header may be matched by a cell's address match cam, which always causes the header to stop.
pl_consume_message_header()	Allows the calling cell to receive a message header from a connection. This action can be performed by any cell along the path of the logical channel that stops the header.
pl_produce_message_trailer()	Allows the calling cell to end a message by sending a message trailer over a connection. Like a message header, a message trailer may be stopped and consumed by any cell along the path of a logical channel. If not stopped by any intermediate cell, the message trailer will flow to the ultimate destination of a logical channel.
pl_consume_message_trailer()	Ends a message on the calling cell by consuming the message trailer.

The consume functions are also provided in forms that cause the header or trailer to be sampled (copied), but not consumed. All of these functions are available in lower overhead forms that select the streamgate via the function called as opposed to the user specifying it as an argument. Refer to Appendix B for a complete description.

Message headers and trailers can be recognized by the communication agent (refer to the description of express in Appendix B). The semantics associated with this capability are up to the user. There are some guidelines that should be observed regarding the use of message headers and trailers, and these are discussed in the section *Using Messages*, later in this chapter.

User Data Movement

PathLib provides streamgate binding operations to support user-programmed send and receives and spooled block data movement. Single-word and double-word streaming, plus support for test of the iWarp condition codes, are provided via the iWarp C compiler intrinsics _sendi(), _sendf(), _sendd(), _receivei(), _receivef(), _received(), _cond(), and _condcall(). Refer to the *iWarp C Manual* for complete descriptions of these intrinsics.

Implicit send and receive operations via the iWarp compute and access instruction are generated through the optimizing iWarp C compiler.

Streaming

Streaming is the user-programmed word-by-word transfer of data over messages by using streamgates. A streamgate is the mechanism that allows the computation agent to access a connection. A streamgate is bound to a connection for input, output, or both. The send and receive operations may be performed to or from the connection by specifying the streamgate.

There are four streamgates: two for inbound operations and two for outbound operations. The streamgates are GATE0_IN, GATE0_OUT , GATE1_IN, and GATE1_OUT . Additionally, streamgates are used by the connections, messages, and marks functions to perform the data production and consumption necessary to implement these operations. The user is expected to supply a streamgate specifier to these functions to identify the gate to be used to perform the operation. For some of these functions, the streamgate binding will be changed as a side-effect of the requested operation.

PathLib includes the following streamgate functions:

pl_bind_input_gate()	Bind an input gate to an inbound connection.
pl_bind_output_gate()	Bind an output gate to an outbound connection.
pl_save_input_binding()	Save the current binding for an input gate in a user-supplied variable.
pl_save_output_binding()	Save the current binding for an output gate in a user supplied variable.
pl_restore_input_binding()	Restore an input gate binding from a user-supplied variable.
pl_restore_output_binding()	Restore an output gate binding from a user-supplied variable.
pl_unbind_input_pct_entry()	Unbind either or both input streamgates from an inbound connection.

Spooling

Spooling allows the iWarp hardware to perform send and receive operations to or from messages and local memory without programmed word-by-word copy operations. Spooling can be thought of as a DMA operation between local memory and logical channels.

PathLib provides a simple interface to the iWarp spooling mechanisms to allow the user to use spooling to send and receive blocks of data.

NOTE

Currently , only word-oriented spooling operations are supported. Double-word spooling is more efficient, but is more difficult to support due to the lack of agreed-upon conventions and protocols to handle the proper alignment of data in local memory . Double-word spooling support is planned for a future release.

PathLib includes the following spool functions:

pl_spool_block_in()	Request the reception of a block of data into local memory from a connection via a spoolgate.
pl_spool_block_out()	Request the production of a block of data from local memory to a logical channel via a spoolgate.
pl_spool_complete()	Determine if a requested spool transfer is complete.

Aggregate Logical Channels (Express Operations)

The user may form aggregate logical channels by plugging connections together to form chains. The connecting cells may selectively participate in communications over the chain. Cells may join and split the connections as necessary to suit the requirements of the application.

When a cell joins connections, data is forwarded from the inbound neighbor to the outbound neighbor. Forwarding is handled by the communication agent, allowing the computation agent to perform other work. Forwarding is stopped by a user-specified sentinel or as a result of the arrival of a matching message header. The various sentinels are message header or trailer, software mark, connection trailer , or when the cell is between two messages on the channel.

PathLib express functions create and destroy aggregate logical channels by providing operations to set the sentinel, forward data, and resume programmed data movement after forwarding has been stopped

PathLib includes the following express functions:

pl_set_sentinel()	Set the sentinel to stop forwarding.
pl_get_sentinel()	Obtain the sentinel in effect.
pl_express_no_mask_toq()	Request that data be forwarded, but do not ignore the sentinel if the first data word to be forwarded satisfies the sentinel.
pl_express_mask_toq()	Request that data be forwarded. Ignore the sentinel if the first data word to be forwarded satisfies the sentinel, and forward until the sentinel is satisfied again.
pl_cancel_requested_express()	Attempt to cancel a previously requested express operation.
pl_stopped()	Determine if forwarding has been stopped.
pl_between_messages()	Determine if the cell is between messages.
pl_synced_with_neighbor()	Determine if data production can be resumed over an outbound connection that was joined to an inbound connection using an express operation.

Software Marks

A software mark is a data word that can be detected by the communication agent. The semantics associated with software marks are defined by the user. An express sentinel can be specified that causes express forwarding to stop when a software mark arrives at a cell. The marks functions allow the user to send, receive and sample software marks.

PathLib includes the following marks functions:

pl_send_software_mark()	Send a software mark.
pl_receive_software_mark()	Receive a software mark.
pl_sample_software_mark()	Copy a software mark.

Support Functions

PathLib also provides functions to initialize an instance of the library linked to a user's program and handle and print errors reported by PathLib functions.

PathLib includes the following initialization functions:

PL_INIT()	Initializes the iWarp hardware for use by PathLib. The user specifies which of the PCT entries and spoolgates available on a cell may be used by the PathLib interface for this instance of the library.
pl_rpe_configure()	Establishes which of the PathLib PCT entries in the calling cells neighbors are to support inbound/express logical channels propagated to them by this cell, and which local PCT entries this cell can use for creating connections.
pl_load_amc()	Load an address match cam and cam mask on this cell.

For a description of the error handling operations provided by PathLib, refer to the description of the plerror functions in Appendix B.

Guidelines for Using PathLib

Because of the flexibility required to use PathLib for program communication on the iWarp array, there are few strictly-enforced policies. While this provides the needed flexibility, it also creates hazards for the programmer.

The following guidelines will help avoid some basic pitfalls involved in designing programs for the iWarp array using PathLib communication.

Cell Naming

Cell naming and addressing are based upon the hardware address-matchcam. Each cell has four matchcams that together control what connection and message headers the cell recognizes.

The user has access to a portion of one cell matchcam. The PathLib function pl_load_amc() provides the user control over the low-order 19 bits of the PathLib matchcam id and mask. Thus, the user has exclusive access to one half of the available match address space. The other half is reserved for the iWarp RTS.

No inter-cell synchronization is provided by PathLib, such as that required to ensure that a receiver's address-matchcam is loaded before a connection reaches the cell. This synchronization is the user's responsibility.

PathLib doesn't address any aspect of how cells are named. This is up to the runtime system and the user. The current RTS gives every cell a canonical name.

Using Connections

To reach its destination, a connection must be granted resources along the route of the logical channel. The propagation of the connection header will stall if it encounters a resource shortage at any cell along its route. If resources become available, the header will then proceed. Once a connection is established, traffic over the connection is guaranteed a share of the available physical bus capacity in all processors along the path of the connection.

The routing of connections is a result of information encoded in the connection header and configuration information installed on the cells. The user has responsibility for correct management of both of these details.

Before connections can be created, proper RPE configuration must be in place within the array. If not properly configured, cells may forward connection headers in unpredictable directions.

Connections may be created that traverse the array in any way the user sees fit. Some important aspects of routing to consider are:

- ensuring that routes are deadlock free
- minimizing the cost of routes in terms of logical channel resources
- maximizing the use of available physical pathway bus capacity of the array
- enhancing the re-usability of cell and array programs by characterizing, consolidating, and standardizing the entry and exit points of inter-processor communications

In particular, the issue of deadlock must be addressed when setting up connections among cells. Due to a lack of synchronization and pathway cleanup primitives, only static configurations of connections should be used.

Assuming static connection allocations, observing the following guidelines will help avoid deadlock situations when setting up connections:

- All cells must have names before connections are created. RTS installs a canonical name for every cell before any user codes may be loaded and run on the array. An alias may be established for a cell through the use of pl_load_amc().

NOTE

> Aliases should not be used for connection setup. They should only be used for messages, and the call to install the alias should be made before any connection setup code is run. There is no inherent synchronization available to the user to ensure that cells have their names in place before communication requiring these names occurs.

- All cells must have established RPE configurations before they begin connection setup. These configurations are established by calling pl_rpe_configure(). The configuration established must provide sufficient resources on all cells to allow all connection setup work to complete.

NOTE

Cells may not change their RPE configurations once they are established without restarting the system.

- Connection setup and data movement over connections must be handled in two distinct and separate steps. First, connection setup should be performed in its entirety. Second, data movement can be performed. Observing this method avoids deadlock that might occur in cases where cells create connections to cells from which they also accept connections.

  For example, if two cells created connections to each other, then started their respective sends before accepting their partner's connection, they would be deadlocked if there isn't sufficient PCT entry queue space between each other to hold the transmissions.

  The simple way to avoid problems of this kind is to program a cell to create all the connections for which it will be the origin, then accept all the connections for which it will be the destination. After both of these steps are complete, data movement may be performed.

- When using pl_accept_connection() when more than one connection is pending at the cell, the connection accepted is arbitrary The user must program any protocol semantics associated with connections. For example, this might involve accepting the connection and processing user-defined protocol messages that follow the header.

- Connections must be terminated using the same cells that created or accepted them.

Using Messages

The communication agent does not require that data transmitted over logical channels be encapsulated by messages. Message headers and trailers may occur arbitrarily , but there is a wide agreement that a more restrictive usage model be employed to allow modular development of array programs, such that independently-developed programs and systems can later be used together.

Observe the following guidelines when using messages:

- Messages should not be nested between any two cells on a logical channel.
- Data words and software marks must not be found outside of a message by any cell participating in a logical channel.
- Cells may produce, consume, and forward (express) message headers and trailers arbitrarily, but between any two cells on a logical channel, data should always be within the context of a message.

  For example, a cell other than the cell that accepted a message header may stop and consume the message trailer for a message. If this is done, the message is complete through the cell that consumes the trailer. If the cell that accepted the header is beyond the cell that consumes the message trailer along the path of a logical channel, the message is closed between the origin of the trailer and the cell that consumed it, but the message remains open from the the cell that intercepted the trailer to the cell that consumed the header . Similarly, if the origin of the trailer is beyond the origin of the header, a message is still open from the cell that was the origin of the header to the cell that is origin of the trailer.

Express

Observe the following guidelines when using express:

- A connection trailer should not be allowed to be forwarded through the cell that is the ultimate destination of the connection.
- A single connection may not be joined to more than one partner at a time.
- Joined connections must be split before streaming or spooling may be resumed over the connections. Downstream neighbor synchronization must be performed before streaming or spooling is resumed on the outbound partner after a split is detected.
- The job of terminating connections that have been members of aggregate logical channels is an array-level activity, not a cell-level activity. Connections are terminated using individual cells, the termination of connections can have array wide consequences, particularly in the case of terminating connections that have been used to form aggregate logical channels. A basic guideline to observe is that aggregate logical channels must be terminated by splitting them into their constituent connections and terminating each of these using the PathLib connections interface.

A PathLib Example

This section contains a simple PathLib example in which a single logical channel is produced and consumed by a cell. The channel is sent out in the Xright direction from the cell, and returns to the cell from the Xleft direction, after traveling around the path of cells that comprise the row of the array in which the cell resides.

The example assumes the following:

- all the cells in the array have names
- this cell's name is provided by a system variable called _cellid
- the systems software has configured cell logical channel resources and has initialized the PL module.

```
#include <pl.h>

int _cellid;

main()
{
  int o_chan;                 /* logical channel descriptor */
  int nr_hdr_words;           /* number of corners to turn to
                                 reach destination */
  int _cellid;                /* name of the destination cell */

  int i_chan;                 /* logical channel descriptor */
  int header;                 /* storage for input header */

  /* Create a logical channel to this in the X-right direction. */
  nr_hdr_words = 1;
```

```
    o_chan = pl_create_connection(GATE0, DIR_XR, &_cellid, nr_hdr_words);
    (void) pl_produce_message_header(GATE0, _cellid);

    /* Accept the logical channel just created. */
    i_chan = pl_accept_connection(GATE0, &header);
    (void) pl_consume_message_header(GATE0);

    /* Send and receive 1000 words over the channel */
    {
      int k;
      for (k=0; k<1000; k++) {
      _sendi(GATE0, k);
      k = _receivei(GATE0);
      }
    }

    /* Terminate the output channel */
    (void) pl_produce_message_trailer(GATE0);
    (void) pl_destroy_connection(o_chan, GATE0);

    /* Terminate the input channel */
    (void) pl_consume_message_trailer(GATE0);
    (void) pl_conclude_connection(i_chan, GATE0);

    exit(0);
}
```

This example provides a simple illustration of programming on the iWarp system using PathLib communication functions. Chapter 5 provides further examples of developing and executing programs on the iWarp system.

PathLib Terms

This section defines terms that relate to PathLib.

aggregate logical channel	A logical channel that is composed of two or more connections, such that data from one connection is automatically forwarded to the other connection. This type of channel is formed via express operations.
connection	An instance of a logical channel created by PathLib. Connections may be joined to form aggregate logical channels that are chains of connections. Cells may also split aggregate logical channels back into its constituent connections (refer to the description of express in Appendix B).
express	1) The state of a logical channel with respect to a cell such that the cell's computation agent does not participate in the movement of data via the channel; 2) To couple an input connection to the RPE of an output connection by putting the input connection's PCT entry into a mode where a cell's communication agent moves data from the input connection to the output connection automatically.

forward	To pass message components through a cell in express mode.
header	A list of control words that specify a route across the array , and that are marked and recognized by the communication agent as being the first component of either a connection or a message.
logical channel	An allocation of resources in the communication agent that forms a pathway through which data may flow from cell to cell. Logical channels always have at least one sender (the producer of the channel) and one receiver (the consumer of the channel).
match	The action performed by the communication agent when a connection or message header control word id field is detected by an address match cam. A match occurs when a matching header reaches the toq and causes transmission to stop, leaving the header at the toq to be sampled or consumed by the computation agent.
mesh	The physical inter-connection of iWarp cells that constitutes an iWarp processor array.
message	A capsule for data carried over a logical channel, the header and trailer of which are defined and recognized by the iWarp communication agent.
PCT	Pathway Control Table; the hardware structure within the iWarp communication agent that stores the cell states of logical channels, including the recipient PCT index, the local and recipient free queue space, and an 8 word data queue.
partner	An input or output connection that is joined to another connection as a result of an express operation.
PCT entry	A record in the PCT that maintains the state for a logical channel.
PCT index	The handle for a specific PCT entry and also the PathLib users handle for a connection; it is an integer defined and understood by the communication agent to designate a PCT entry.
RPE	Recipient PCT Entry: a PCT entry referenced from an adjacent cell that is a downstream component of a logical channel.
RPI	A Recipient PCT Entry index; similar to a PCT entry index but is a reference to an entry on an adjacent cell.
routing destination	A cell that matches a connection header and is instructed by the header to select a new physical pathway upon which to propagate the connection header .

spoolgate	Any of eight physical interfaces between the computation agent and the communication agent through which spool transfer operations may occur.
spool index	The PathLib programmers handle on a spoolgate.
spooling	The transmittal and reception of blocks of data to and from local memory concurrent to, and without direct aid from, a program running on the cell using spoolgates. Spooling allows the PathLib programmer to initiate the movement of data, then have the cell move data to or from a logical channel and local memory without further programmed assistance.
stopped	The state of a logical channel with respect to a cell such that the cell's communication agent is not automatically forwarding data over the channel; not express.
streamgate	Any of four physical interfaces between the computation agent and the communication agent through which the computation agent may send and receive data over logical channels under direct program control.
streaming	The programmed word-by-word transmittal and reception of data over streamgates using streamgates.
toq	The record in the PCT entry data buffer that contains the next word to be processed by the computation agent or scheduled for transmission by the communication agent. Usually, the toq is only relevant with respect to the processing of inbound or express PCT entries, corresponding to the local PCT entry for an inbound connection or express partner.
trailer	The last word of either a connection or a message.
ultimate destination	The cell that accepted a connection via pl_accept_connection(), or the last cell on an aggregate logical channel.

5 Developing a Program

Introduction

This chapter provides an introduction to developing and executing programs for the iWarp system. Because parallel programs are highly application specific, it is impossible to provide a cookbook recipe for parallel program development. This chapter simply provides some basic examples to illustrate the fundamentals of program development and execution. These examples include

- a single-cell ‘‘Hello World” program
- a four-cell ‘‘Hello World” program
- a single-cell program using PathLib communication
- a matrix multiplication program using PathLib communication

A Single-Cell Example: “Hello World”

The following example is a simple single-cell example that illustrates the steps involved in developing and executing a program for the iWarp system. The example program is the following ‘‘Hello World” program:

```
#include <stdio.h>

main()
{
  printf(``hello world\n");
}
```

Assuming the program is stored in the file hello.c, use the following command to compile the program and store the output in the file hello.out:

```
iwcc -o hello.out  hello.c
```

Chapter 2 provides a complete description of the iwcc command.

Before you can load and execute the compiled program, you must link the program to the RTS software with the rtslink command as follows:

```
rtslink -o hello.run  hello.out
```

Chapter 3 provides a complete description of the rtslink command.

You now have an image that is ready to load and execute stored in the file hello.run. Now you must locate an array for program execution. Array names have the following form:

```
/dev/isibn
```

where *n* is the array number. Configuration information about each array is shown in the configuration file, /iwarp/etc/config. The following is an example of a /iwarp/etc/config file:

```
#
#  @(#)config   230.4 6/27/91 - Configuration File Change Control
#
#  This file lists each iWarp system available at this site
#
#  Each line is interpreted as follows:
#    starts with '#': comment - ignored
#    starts with '%': followed by hostname and SIB #
#    other: a keyword followed by the information for that keyword
#           keyword lines apply to the last named iWarp (last '%' line)
#           if no '%' line has occurred yet, the lines are ignored
#           lines which are REQUIRED for each machine:
#            "CELLS #"- number of array cells attached
#            "WIDTH #" and "HEIGHT #" - dimensions of the array
#            lines which are OPTIONAL for each machine:
#            "NODAK" - DAK is not present on the SIB
#            "SIBONLY" - no array is attached to the SIB (be sure to still
#                          enter CELLS, WIDTH and HEIGHT as all zero)
#
#  read via "/iwarp/bin/iwconfig" and "iwconfig()"
#

% stargazer 0
CELLS 4
WIDTH 2
HEIGHT 2
SYSHZ 16000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE 0:0xe 7:0xd 64:0x0

% stargazer 1
CELLS 8
WIDTH 4
HEIGHT 2
SYSHZ 10000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE
#0:0xe 1:0xd 4:0x0

#% stargazer 2
CELLS 64
WIDTH 8
HEIGHT 8
SYSHZ 10000000
LMSIZE 0x80000
ARCH 3.2
NEIGHBORMODE
```

The configuration file shows that there are three iWarp arrays. The array on /dev/isib0 is a 2 by 2, the array on /dev/isib1 is a 4 by 2, and the array on /dev/isib2 is an 8 by 8.

Before loading a program onto an array for execution, it is advisable to lock the array . If you are loading and executing the array by hand with iwmon, you must lock and unlock the array yourself. If you are using lgo to load and execute your program, lgo locks and unlocks the array for you. If you were going to load and execute your program with iwmon on /dev/isib0, you would lock the array with the following command:

```
siblock 0
```

If the lock is successful, you will get the following message:

```
SIB 0 locked (count = 1)
```

If someone else is using the array, you will get a message like the following:

```
Unable to lock SIB 0:
-rw-rw-rw-   1 hof           146 Mar 27 11:10  /usr/tmp/sib.0
---
Locked count:  2
Effective User ID:    hof
Real User ID:         iwarp!hof     ttyp9   Mar 26  14:59
Time of last lock:    Tue Mar 26  15:46:02 PST  1991
```

If you wanted to load and execute the single-cell program on cell 0 of the array, you could do so with the following lgo command:

```
lgo -r hello.run 0 -w 1 /dev/isib0
```

Chapter 3 provides a complete description of the lgo command.

When you execute the program, your output will be like the following:

iWarp Monitor V2.1 Mon July 1 14:20:24 PST 1991 coff V2.1

```
resetting the array...
booting/iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991  hello world
0 Mar 27 13:25:54 1991 exit: status=0000000c
```

As can be seen from the output, lgo calls the iWarp monitor to load and run the program. You can also use the monitor to load, run, and debug your program interactively . The monitor and monitor commands are described in Chapter 3.

The output shows that any output that comes to the terminal is tagged with the id of the cell and the date as seen in the line:

```
iwmon> 0 mar 27 13:25:54 1991  hello world
```

When the program exits, the exit status is printed. Our “hello world” program did not call exit, so our exit status is not meaningful.

After running the program, you should unlock the array for use by others. To unlock /dev/isib0, use the sibunlock command as follows:

```
sibunlock 0
```

"Hello World" on Four Cells

The ''hello world" program for four cells would be the following:

```
#include <stdio.h>

int _cellid;

main()
{
    fprintf(stdout, "hello world from cell %d\n", _cellid);
    exit(0);
}
```

In this program we've added a call to exit and a variable called _cellid. This variable receives the logical cell id of a cell from the runtime system. To compile, link, and run the program on 4 cells on the array on /dev/isib0, enter the following commands (% represents the Sun prompt):

```
% iwcc -o hello.out hello.c

% rtslink -o hello.run hello.out

% lgo -r hello.run 0 1 2 3 -w 4 /dev/isib0
```

iWarp Monitor V2.1

Mon July 1 14:20:24 PST 1991

coff V2.1

```
resetting the array...
booting /iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991  hello world from cell 0
2 mar 27 13:25:54 1991  hello world from cell 2
3 mar 27 13:25:54 1991  hello world from cell 3
1 mar 27 13:25:54 1991  hello world from cell 1
iwmon> %
```

A Single-Cell Program Using PathLib Communication

The examples shown so far have illustrated the mechanics of executing programs on an iWarp array, but they have not discussed the most challenging aspect of parallel programming on the iWarp system; program communication between cells. Chapter Four provided an introduction to the PathLib library of communication functions. This library provides functions to implement program-based communication in programs developed for the iWarp system. Appendix B also provides descriptions of the PathLib functions.

This section provides an example of a single-cell program that uses PathLib functions to communicate with itself. In this example we are assuming a two by two array , configured as shown in Figure 5-1.

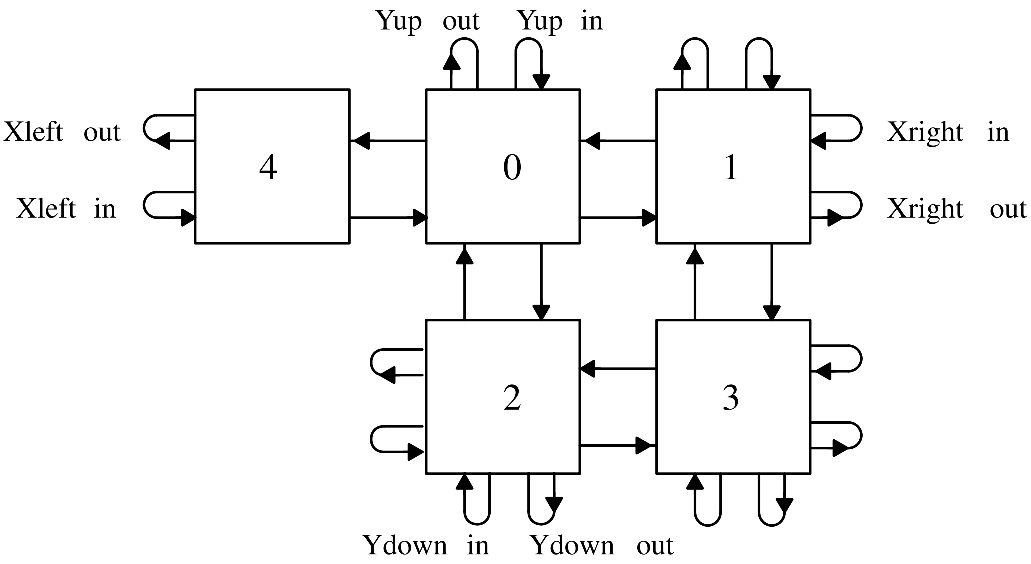


Figure 5-1: Two by Two Array Configuration

Note that cell 4 in the configuration is the interface cell, and cells 0 through 3 are the array cells. In general, with an N-cell system, the array cells are numbered 0 through N–1, and the interface cell is numbered N.

For this example, the program is loaded and run on cell 0. Cell 0 sends out a channel to the right, and the channel returns to cell 0 on the left after passing through cell 1 and cell 4. The following is the example program.

```
#include <stdio.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>

int _cellid;
```

```
main()
{
  int o_chan;                   /*  logical channel descriptor */
  int nr_corners;               /*  number of corners to turn to
                          reach destination */
  int dest_id;                  /*  name of the destination cell */

  int i_chan;                   /*  logical channel descriptor */
  int header;                   /*  storage for input header */
  int i;
  int tmp;

  /* Initialize the PL module */
  PL_INIT(_cellid, 0xfffc, 0xff);

  /* Set up PCTs for neighbors */
  pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);

  /* Assume the program is to run on cell 0 only */
  if (_cellid != 0) {
    exit(0);
  }

  /* Create a logical channel to this cell in the Xright direction */
  dest_id = _cellid;
  nr_corners = 1;

  o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR, &dest_id,
                                    nr_corners);
  (void) pl_produce_message_header(GATE0_OUT,dest_id);

  /* Accept the logical channel just created. */
  i_chan = pl_accept_connection(GATE1_IN, &header);
  (void) pl_consume_message_header(GATE1_IN,&header);

  /* Send and receive 1000 words over the channel */
    for (i=0; i<1000; i++) {
      _sendi(GATE0_OUT, i);
      tmp = _receivei(GATE1_IN);
      if (tmp != i) }
        fprintf(stderr, "error: sent %d, recd %d\n", i, tmp);
        exit(1);
      }
    }

  /* Terminate the output channel */
  (void) pl_produce_message_trailer(GATE0_OUT);
  (void) pl_destroy_connection(o_chan,GATE0_OUT);

  /* Terminate the input channel */
  (void) pl_consume_message_trailer(GATE1_IN);
  (void) pl_conclude_connection(i_chan,GATE1_IN);

  exit(0);
}
```

A section by section discussion of the example is helpful to understanding how PathLib communication works.

```
/* Initialize the PL module */
PL_INIT(_cellid, 0xfffc, 0xff);
```

PL_INIT initializes the PathLib module. PL_INIT told PathLib that it could access 18 of the 20 PCTs, with the argument 0xfffc defining which ones. The two PCTs allocated to iwmon, PCT0 and PCT1 are not available for use by PathLib. PL_INIT also tells PathLib which spoolgates are to be used. There is a maximum of eight, and in this example the argument 0xff is a bit mask indicating that all of the spoolgates are to be used.

```
/* Set up PCTs for neighbors */
pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);
```

pl_rpe_configure estalishes the allocation of logical channel resources for neighbor cells. The first four arguments to the function represent RPEs for the four directions, Xleft, Xright, Yup, and Ydown. The last argument defines which RPE is used for output connections. The arguments to pl_rpe_configure are bit masks. representing which RPEs to use. There are a total of 20 RPEs, numbered 0 through 19. Examining the arguments we see that for this example:

- Xleft uses RPEs 4 and 5
- Xright uses RPEs 6 and 7
- Yup uses RPEs 8 and 9
- Ydown uses RPEs 10 and 11
- RPEs 12-15 are used for output connections.

Note that RPEs 0 and 1 are reserved for the monitor. RPEs 2, 3, and 16-19 are not used in this example.

```
/* Create a logical channel to this cell in the Xright direction */
dest_id = _cellid;
nr_corners = 1;

o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR, &dest_id,
                                        nr_corners);
(void) pl_produce_message_header(GATE0_OUT,dest_id);
```

pl_create_connection creates a connection bound to a stream gate routed in an initial direction with routing information. Looking at each of the arguments:

GATE0_OUT	A stream gate. There are two gates available, GATE0 and GATE1. Each gate has an input and an output side.When you write (send) to the output, the data is written out to the logical channel to which the output side of the gate is bound. When you read from the input side, data is read (receive) from the logical channel to which the input side of the gate is bound.
PL_DIR_XR	Specifies the initial direction of the connection, in this case, Xright. This argument determines the direction of the first routing control word.
dest_id	an array of routing control words.
nr_corners	How many routing control words are in the array.

pl_produce_message_header sends a message header via the outbound connection. GATE0_OUT is the streamgate bound to the output PCT entry that is the origin of the connection. dest_id specifies the cell address.

We have created a connection to ourself and have sent a message, now we must accept the message.

```
/* Accept the logical channel just created. */
i_chan = pl_accept_connection(GATE1_IN, &header);
(void) pl_consume_message_header(GATE1_IN,&header);
```

pl_accept_connection consumes the header for the connection and returns it in i_chan. pl_consume_message_header consumes the message header. We are now ready to send and receive data.

```
/* Send and receive 1000 words over the channel */
  for (i=0; i<1000; i++) {
    _sendi(GATE0_OUT, i);
    tmp = _receivei(GATE1_IN);
    if (tmp != i) }
      fprintf(stderr, "error: sent %d, recd %d\n", i, tmp);
      exit(1);
    }
  }
```

In the for loop, we send data from GATE0_OUT , and receive from GATE1_IN. When the for loop is complete, we must clean up the connections.

```
/* Terminate the output channel */
(void) pl_produce_message_trailer(GATE0_OUT);
(void) pl_destroy_connection(o_chan,GATE0_OUT);
```

pl_produce_message_trailer closes the message, and pl_destroy_connection closes the output connection.

```
/* Terminate the input channel */
(void) pl_consume_message_trailer(GATE1_IN);
(void) pl_conclude_connection(i_chan,GATE1_IN);
```

pl_consume_message_trailer consumes the message trailer and pl_conclude_connection closes the input connection.

We now compile the program:

```
iwcc -o example.out example.c
```

Since we are going to run the program on the interface cell as well as the array cells, we have to do two seperate links to the runtime system:

```
rtslink -sib -o example.sib.run example.out
```

```
rtslink -o example.run example.out
```

Next, we generate a run script with lgo:

```
lgo -n -o lgo.script -R example.sib.run 4 -r example.run 0..3 /dev/isib0
```

This command generates the following file, lgo.script:

```
load 4 example.sib.run
run 4 0x20000
load 0 example.run
run 0 0x12000
load 1 example.run
run 1 0x12000
load 2 example.run
run 2 0x12000
load 3 example.run
run 3 0x12000
exit
```

In this example, it is critical that all cells have called PL_INIT and pl_rpe_configure before cell 0 starts processing. To do this, we can edit lgo.script to ensure that cells 4, 1, 2, and 3 have started and exited before cell 0 starts. The following is the edited lgo.script:

```
load 4 example.sib.run
run 4 0x20000
load 1 example.run
run 1 0x12000
load 2 example.run
run 2 0x12000
load 3 example.run
run 3 0x12000
wait 4
load 0 example.run
run 0 0x12000
wait 1
exit
```

We can now load and execute the program by feeding lgo.script back into lgo as follows:

```
lgo -f lgo.script /dev/isib0
```

This example illustrates the importance of inter-cell synchronization. In this case it was easy to synchronize the cells, but that is not always the case. There is no general mechanism to handle inter–cell synchronization, and it must be thought out on an application by application basis.

Matrix Multiplication Using PathLib Communication

The following example solves a matrix multiplication on a two by two iWarp array. The matrices are four by four. Figure 5-2 shows the contents of matrix A and B, and the desired contents of the result matrix, matrix C.

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}} \text{X} \underset{\text{Matrix B}}{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} 10 & 10 & 10 & 10 \\ 26 & 26 & 26 & 26 \\ 42 & 42 & 42 & 42 \\ 58 & 58 & 58 & 58 \end{bmatrix}}$$

Figure 5-2: Contents of Matrices for Matrix Multiplication

Our strategy is to distribute one row of matrix A to each cell in the array. Each cell can then compute a row of the result matrix. When each cell finishes computing, the result rows will be collected to compile the result. Figure 5-3 shows our four-cell configuration.

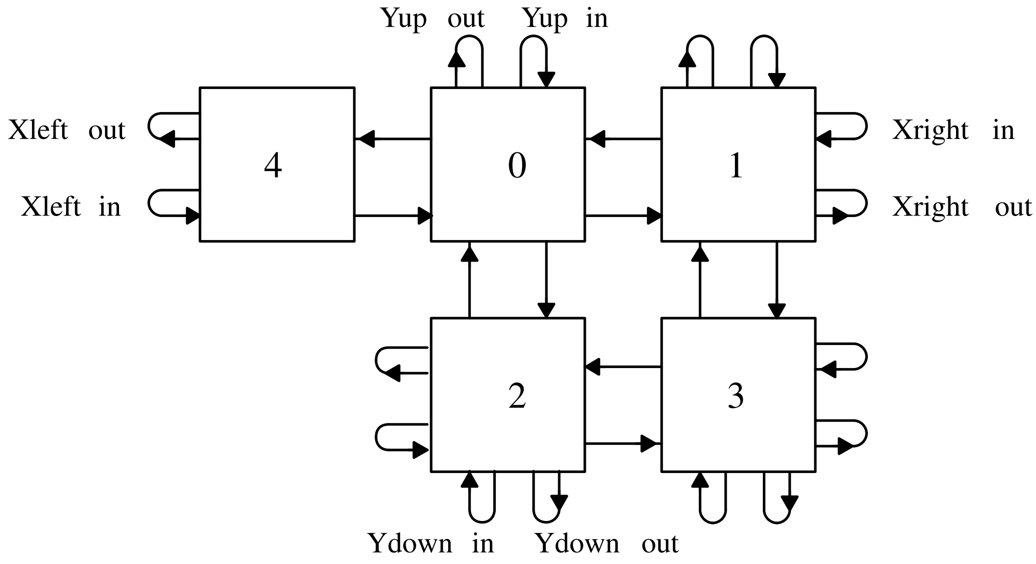


Figure 5-3: Two by Two Array Configuration

We will compile the result matrix on cell 0, using the following paths to return the results from the other cells:

- cell 1: Xleft to cell 0
- cell 2: Yup to cell 1, Xleft to cell 0
- cell 2: Xright to cell 3, Yup to cell 1, Xleft to cell 0

The following is the program.

```
#include <stdio.h>
#include <iwarp_cc.h>
#include <gates.h>
#include <regnums.h>
#include <asm/gen_asm.h>
#include <asm/pw_asm.h>
#include <pathlib/pl.h>

int mata[4][4] = {
      1,  2,  3,  4,
      5,  6,  7,  8,
      9, 10, 11, 12
     13, 14, 15, 16,
};

int matb[4][4] = {
      1, 1, 1, 1,
      1, 1, 1, 1,
      1, 1, 1, 1
      1, 1, 1, 1,
};

int matc[4][4] = {
      0, 0, 0, 0,
      0, 0, 0, 0,
      0, 0, 0, 0
      0, 0, 0, 0,
};

int     _cellid;

void    do_row();
void    collect_results();
void    send_results();

main()
{
  int i;
  int j;

  /* Initialize the PL module */
  PL_INIT(_cellid, 0xfffc, 0xff);

  /* Set up PCTs for neighbors */
  pl_rpe_configure(0x0030, 0x00c0, 0x0300, 0x0c00, 0xf000);

  /* Distribute Matrix A */
  do_row(_cellid);

  if (_cellid == 0) {
    collect_results();
    fprintf(stdout, "\nresult of matrix A * matrix B\n\n"
    for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {

        fprintf(stdout, "%2d ", matc[i][j]);
      }
```

```
      fprintf(stdout, "\n");
    }
    fprintf(stdout, "\n");
  }
  else {
    send_results();
  }

  exit(0);


void
do_row(row)
  int row;
{

  int col;
  int i;
  int sum;
  for (col=0; col<4; col++) {
    sum = 0;
    for (i=0; i<4; i++) {
      sum += mata[row][i] * matb[i][col];
    }
    matc[row][col] = sum;
  }
}



void
collect_results()
{

  int i_chan;
  int header;
  int fromid;
  int i;
  int j;

  for (i=1; i<4; i++) {


    /* Accept the logical channel */
    i_chan = pl_accept_connection(GATE1_IN, &header);
    (void) pl_consume_message_header(GATE1_IN, &header);

    /* First data word in cellid of sender */
    fromid = _receivei(GATE1_IN);

    for (j=0; j<4; j++) {
      matc[fromid][j] = _receivei(GATE1_IN);
    }
```

```
    /* Terminate the input channel */
    (void)pl_consume_message_trailer(GATE1_IN);
    (void)pl_conclude_connection(i_chan,GATE1_IN);
  }
}


  /* cell 1: xl to cell 0
     cell 3: yu to cell 1, xl to cell 0
     cell 2: xr to cell 3, yu to cell 1, xl to cell 0  */

void
send_results()
{
  int o_chan;
  int headers[4];
  int header_count;
  int i;

  switch (_cellid) {

  case 1:
      /*
      * asm_build_header
      *      dest:   0
      *      turn:   0=no, 1=yes
      *      pw:     0=X, 1=Y
      *      dir:    0=left, 1=right
      *      pool:   0, 1
      */
      headers[0] = asm_build_header(0, 0, 0, 0, 0);
      header_count = 1;
      o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XL,
                                    headers, header_count);
      break;

  case 3:
      headers[0] = asm_build_header(1, 1, 0, 0, 0);
      headers[1] = asm_build_header(0, 0, 0, 0, 0);
      header_count = 2;
      o_chan = pl_create_connection(GATE0_OUT, PL_DIR_YU,
                                    headers, header_count);
      break;

  case 2:
      headers[0] = asm_build_header(3, 1, 1, 1, 0);
      headers[1] = asm_build_header(1, 1, 0, 0, 0);
      headers[2] = asm_build_header(0, 0, 0, 0, 0);
      o_chan = pl_create_connection(GATE0_OUT, PL_DIR_XR,
                                    headers, header_count);
      break;
  }
```

```
  (void)pl_produce_message_header(GATE0_OUT,0);

  /* Let receiver know what cell data is coming from */

  _sendi(GATE0_OUT, _cellid);

  /* Send result row */

  for (i=0; i<4; i++) {
    _sendi(GATE0_OUT,matc[_cellid][i]);
  }

  /* Terminate the output channel */

  (void)pl_produce_message_trailer(GATE0_OUT);
  (void)pl_destroy_connection(o_chan,GATE0_OUT);
}
```

As with our previous example, we compile, link to the RTS, and generate a command script for program execution.

```
iwcc -o matmult.out matmult.c

rtslink -o matmult.run matmult.out

lgo -n -o matmult.script -R matmult.sib.run 4 -r matmult.run 0..3 \
                               /dev/isib1
```

The output of the run script generated by lgo is the following:

```
load 4 matmult.sib.run
run 4 0x20000
load 0 matmult.run
run 0 0x12000
load 1 matmult.run
run 1 0x12000
load 2 matmult.run
run 2 0x12000
load 3 matmult.run
run 3 0x12000
exit
```

We must edit the script as follows to provide the correct starting synchronization for the cells:

```
load 4 matmult.sib.run
run 4 0x20000
load 0 matmult.run
run 0 0x12000
load 1 matmult.run
run 1 0x12000
load 3 matmult.run
run 3 0x12000
load 2 matmult.run
run 2 0x12000
wait 5
exit
```

Now we can load and execute the program.

```
%lgo -f matmult.script /dev/isib1
```

iWarp Monitor V2.1

Mon July 1 14:20:24 PST 1991

coff V2.1

```
resetting the array...
booting /iwarp/lib2.1/iwmon/package.4...
iwmon> 0 mar 27 13:25:54 1991
result of matrix A * matrix B

  0 Mar 27 13:25:54 1991  10 10 10 10
  0 Mar 27 13:25:54 1991  26 26 26 26
  0 Mar 27 13:25:54 1991  42 42 42 42
  0 Mar 27 13:25:54 1991  58 58 58 58
  0 Mar 27 13:25:54 1991
iwmon> %
```