The Diamond User Guide is now an online document.

It can be found here:

http://help.3l.com/Sundance/

An older version of the user guide follows this page.

Diamond User Guide

Sundance Edition V3.0

Diamond User Guide : Sundance Edition V3.0

Published 1 June 2005

Table of Contents

Part I. Tutorial

An Introduction to Diamond

Table of Contents

Chapter 1. Introduction

Intended Audience

This User Guide accompanies the TI C6000 edition of 3L's real-time operating system, Diamond. It is intended for anyone who wants to program a C6000 system, whether writing a conventional sequential program or using the full support for multi-processor networks and concurrency which Diamond has to offer. It describes how to build Diamond applications at the MS-DOS prompt line and run them with the Windows Server.

Development Environments

Diamond for the C6000 is available on PCs running:

- Windows 2000
- Windows NT
- Windows XP

This Guide assumes that you are reasonably familiar with the operating system of the host computer being used. Some DSP hardware may work on only a limited range of operating systems.

Further Reading

Texas Instruments' publications that may be useful include:

- TMS320C62x/C67x Programmer's Guide
- TMS3206x Optimizing C Compiler User's Guide
- TMS320C6x Assembly Language Tools User's Guide

Conventions

Words, Bytes and Octets

The term "**`word`**" will generally be used to mean a 32-bit data item. We refer to 8-bit items as "**`octets`**" .

Users of Diamond on other processors, including notably the Texas Instruments TMS320C4x, should note that the smallest addressable unit of memory on the C6000 is an octet. This corresponds to C variables of type char or unsigned char. Strict conformity with the ANSI standard mandates the term byte for items of this size, and we shall occasionally use it. However, when we do so, we will make explicit what size of data item we mean, in order to avoid confusion for users of other processors.

Links, Ports, and Wires

Diamond on the C6000 uses the same communication model as the implementations on other processors, including the C4x. However, at the hardware level, C6000 boards use a wide variety of techniques to carry out communications between processors. No matter what techniques are used in a particular case, Diamond sees each processor as possessing a number of communication links. Communication takes place between these links, and we shall refer to the connections between them as wires. Regardless of the underlying hardware, you can use Diamond's links and wires in the same way.

The word port, unadorned, has a different meaning, which is discussed in Ports and Channels.

DMA

The term "**DMA**" will be used in this Guide to refer to both the older DMA and the newer EDMA mechanisms.

Typefaces

Throughout this manual, text printed in this typeface represents direct verbatim communication with the computer: for example, pieces of C program, or commands to the host operating system.

In examples, text printed in this typeface is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of the command for running a program:

```
3L x bootable-file
```

This means that the command consists of:

- The words 3L x typed exactly like that.
- A bootable-file: not the text bootable-file, but an item of the bootable-file class, for example myprog.app.

Warning

Information that is intended to draw your attention to potential errors, unexpected behaviour, or facilities that should only be used in exceptional circumstances will be displayed like this.

Examples

When an example of a command is reproduced verbatim, we shall represent the prompt with the sign ►

For example: ► 3L x hello.app

Specialised Information

Information in this colour contains details unlikely to be of direct interest to many users of Diamond. Further details on the points covered are available from 3L.

Installation Folder

The installation directory is where Diamond and its associated software will have been installed. It is possible to install Diamond in any directory, but this document will assume you have used the default directory `C:\3L\Diamond\`. It is best to reserve a directory just for this purpose; do not try to install into the same directory as another product, whether from 3L or any other source. However, you should use the same directory for all editions of Diamond (for example, c6000 and Power PC). Doing this will simplify building applications for all classes of processor.

This document will use <Diamond> to refer to the installation directory.

Support Services

After-sales support is available from your dealer, and also directly from 3L. Support agreements can be purchased for long term or intensive support, and include a free upgrade service.

You can email questions to 3L [mailto:support@3L.com]

Further information is usually available on our web site: http://www.3l.com

Alternatively, you can call +44 131 620 2641, from 9am to 5pm UK time.

Sample Code

Source code for the examples mentioned in the text can be found in the the Diamond installation directory `<Diamond>\target\c6000\Sundance\examples\general`. Complete example programs can be found in subdirectories called `<Diamond>target\c6000\Sundance\examples\Example-n`.

Acknowledgements

The network communication tasks which form a part of this product are based upon the UPR/VCR technology developed by Mark Debbage, Mark Hill and Denis Nicole of the Department of Electronics and Computer Science at the University of Southampton, England. We are happy to acknowledge the assistance we have received from the developers of this technology.

Chapter 2. Sundance Installation

This chapter contains instructions for setting up Diamond after it has been installed from the supplied media, and making it ready for use on Sundance hardware. There is also a brief confidence-test procedure, which you can use to check that the software has been installed correctly.

If you are unfamiliar with Diamond, you may wish to start by reading the Getting Started chapter.

Prerequisites

Before running the software:

- Check that you have the correct edition of Diamond for your target hardware;
- Make sure that your C6000 board has been correctly installed and set up, as described in the manufacturer's documentation;
- Ensure that the Texas Instruments code generation tools are installed. These usually come as part of TI's Code Composer Studio product [SPRU509]. This version of Diamond requires version 5 (or later) of the TI compiler and linker; these are shipped with CCS version 3, or later.

Installed Libraries

The installation processes will have placed a number of libraries in the system folder of your PC, commonly C:\WINNT\SYSTEM32. These are:

Serve3L.DLL	The main server module
3LLnkMan.DLL	The server’s link manager
3LOptMan.DLL	The server’s options manager

Device Driver Software

The Diamond host server communicates with the hardware via a driver that is supplied by your hardware vendor.

Code Composer Studio

You must install the Texas Instruments C6000 code generation tools (C compiler, linker, etc.) before you can use Diamond. The code generation tools are installed as part of the Code Composer Studio product. Install this as described in the accompanying Texas Instruments documentation. Diamond applications are built using commands that call the TI tools from a Command Prompt.

Code Composer Driver Software

The Code Composer JTAG debugger supports a range of different hardware platforms via plug-in device drivers. If you plan to use the debugging features of Code Composer, you must install the appropriate drivers, available from your hardware supplier.

Environment Variables

The installation procedure will have defined some environmental variables for you. You should check their values from the Windows control panel using System/Advanced/Environment Variables.

PATH	defines the DOS search path. It must include the Diamond executable folder, normally C:\3L\Diamond\bin. In general, it is best to put this as near the beginning as you can, so that it takes precedence over any previously installed software.
C6X_C_DIR	defines the default location for the C compiler's #include files and the linker's object libraries. This should have been set up by the installation of the TI compiler. You should not change this otherwise the compiler may pick up the wrong #include files and your programs will not link correctly.

Other processors

The examples in this guide assume that you will be building applications for generic C6000 processors. Diamond includes support for tasks that will run on other types of processor:

C4x Support

Diamond will support applications developed for C4x processors using 3L's Parallel C for the C4x. The server will run applications that have a C4x processor as the root if you select the C4x option. Note that this option must not be selected if you have a C6x processor as the root of your network, even if there are C4x processors elsewhere.

C67 Support

Diamond includes support for the C67xx floating-point DSPs. They are treated identically to the fixed-point C6000 processors, except for compiling and linking, which use variants of the 3L command:

	C6000	**C67xx**
Compiling	3L c hello.c.	3L c67 hello.c
Linking (full)	3L t hello	3L t67 hello
Linking (stand alone)	3L ta hello	3L t67a hello

The **`3L c67`** command invokes the C compiler's mv6700 switch. 3L t and 3L ta use C6700 versions of the libraries.

Configuration (**`3L a`**) and execution (**`3L x`**) are the same for both fixed and floating-point board variants. In particular, the same processor TYPE names are used in **`.cfg`** files.

C64 Support

Diamond includes support for the C6400 family of DSPs. They are treated identically to the fixed-point C6000 processors, except for compiling and linking, which use variants of the 3L command:

	C6000	**C64xx**
Compiling	3L c hello.c.	3L c64 hello.c
Linking (full)	3L t hello	3L t64 hello
Linking (stand alone)	3L ta hello	3L t64a hello

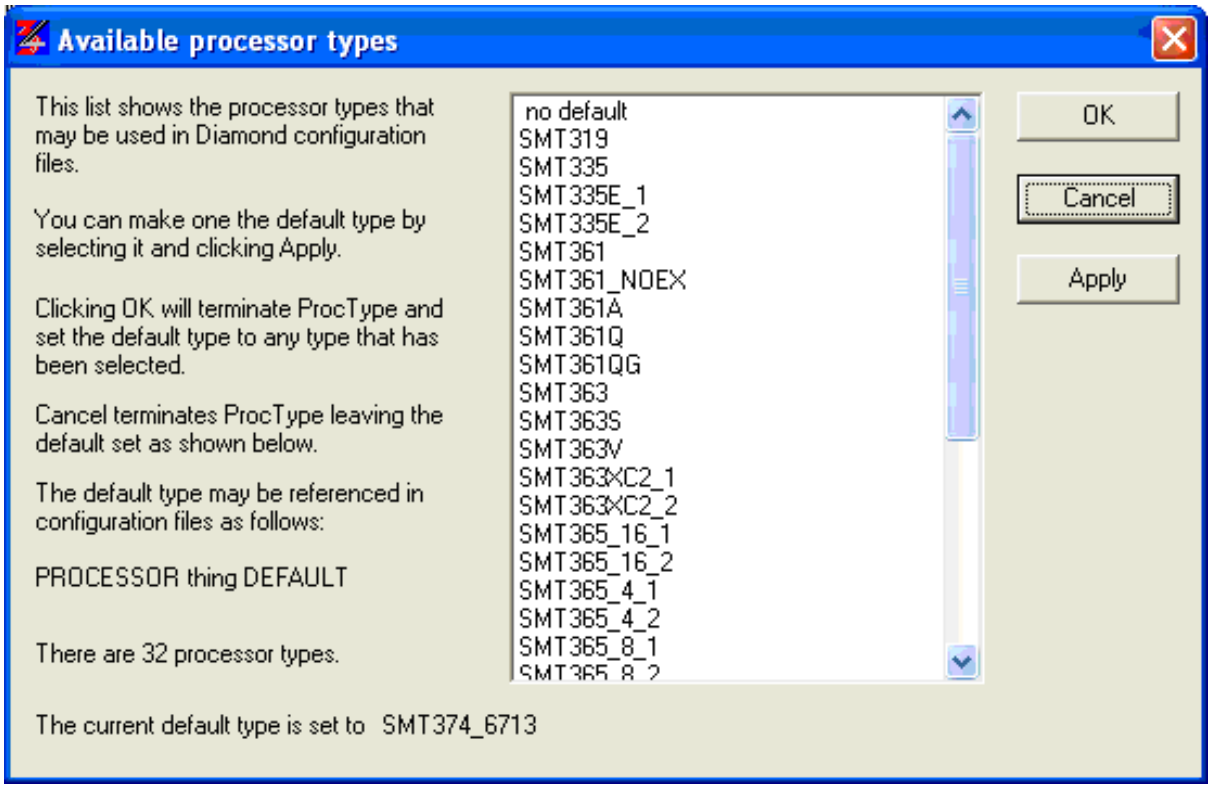
The **`3L c64`** command invokes the C compiler's **`mv6400`** switch. **`3L t64`** and **`3L t64a`** use C6400 versions of the libraries.

Configuration (**`3L a`**) and execution (**`3L x`**) are the same for both fixed and floating-point board variants. In particular, the same processor TYPE names are used in **`.cfg`** files.

Available Processor Types

The Sundance edition of Diamond supports a large number of different C6000 modules. The particular modules that you are using in an application are identified by processor types that you use when defining your processors in configuration files.

The Diamond utility program ProcType will generate a list of all the processor types you can use.



To simplify the case where a particular type of module is used frequently, ProcType allows you to set one to be the default processor type, DEFAULT. The installation procedure will offer to set a default processor type for you. The examples provided with Diamond all use this default processor type so that you do not have to change them to match your particular hardware configuration.

Processor types are used in configuration files as follows:

```
PROCESSOR root  SMT361      ! explicit processor type SMT361
PROCESSOR node DEFAULT      ! current default processor type
```

Multiple-processor TIMs

It is important to distinguish between processors and TIMs when using Diamond with Sundance hardware. TIMs, such as the SMT361, that have only a single processor do not cause any problems; Diamond uses the TIM name in configuration files to refer to the type of processor being declared.

```
PROCESSOR root SMT361
```

You need to be more careful when dealing with multiple-processor TIMs, such as the SMT361Q or the SMT374. Once again, Diamond uses the TIM name to identify the type of processor being declared, but it identifies only *one* of the available processors on the TIM. You need a separate PROCESSOR statement for each processor you wish to use. For example, an SMT361Q [1] TIM has four processors, and they can be identified as follows:

```
PROCESSOR DSP_A SMT361Q
PROCESSOR DSP_B SMT361Q
PROCESSOR DSP_C SMT361Q
PROCESSOR DSP_D SMT361Q

WIRE  ?  DSP_A[1]   DSP_B[4]
WIRE  ?  DSP_A[4]   DSP_C[1]
WIRE  ?  DSP_A[0]   DSP_D[3]
WIRE  ?  DSP_B[3]   DSP_C[0]
WIRE  ?  DSP_B[4]   DSP_D[1]
WIRE  ?  DSP_C[3]   DSP_D[0]
```

Note

Processor names do not identify physical processors; the WIRE statements do this. For example, the processor being called DSP_B is the one we get to from the processor called DSP_A by going out of comport 1. Also note that every Diamond application must have one processor called "root". If an SMT361Q is the only TIM in your system, it would need to be declared as:

```
PROCESSOR root  SMT361Q
PROCESSOR DSP_B SMT361Q
PROCESSOR DSP_C SMT361Q
PROCESSOR DSP_D SMT361Q
```

[1] For historical reasons, the processors on an SMT361Q can also be identified by the type "SMT361_NOEX".

Multi-Processor Systems

Diamond's physical links 0–5 correspond directly to Sundance comports 0–5. Note that on some TIMs (for example, the SMT376) not all of these six comports exist; the missing comports behave as though the other end of the link never responds. Comport numbers are used in the configuration file's WIRE statements. The folder `<Diamond>target\c6000\Sundance\examples\example-3` contains the file DUAL.CFG, which shows an outline of the configuration statements required to take the two tasks used in UPC.CFG and configure them to run on two separate processors. You must fill in the correct processor TYPE attributes for the TIMs you are using and modify the WIRE statement to match the cabling on your carrier board. The example has:

```
wire ? root[1] node[4]
```

This tells the configurer to assume that comport number 1 of the root (TIM site 1) processor is connected to comport 4 of the TIM site containing the node processor. You must ensure that the corresponding comport cable is fitted between connectors T1C1 and T2C4 (assuming the node processor is in TIM site 2). If the required comport cable is not physically present, the Diamond host server will not be able to load the configured application. Some Sundance carrier boards can make this connection using built-in switches. See the server's option Board/Properties to change these switches.

Mixed-Processor Systems

Some hardware allows systems to be constructed from a mixture of C6000 and C4x TIMs. Diamond supports this feature in software. To build a mixed C6000/C4x application, you will need:

- Diamond for the C6000; and
- Single (03319) or multi-processor (03308) Diamond for the C4x. Either will do, even if you have multiple C4x and C6000 processors.

The following example shows how to build a version of the UPC example where the root processor is a C6000 and the node is a C40:

```
► 3L c      driver.c
► 3L t      driver
► c40c      upc
► c40cslink upc
► 3L a      mixed.cfg mixed.app -a
► 3L x      mixed
```

The configuration file, MIXED.CFG, will be the same as DUAL.CFG, except for the two PROCESSOR statements:

```
processor root type=myC6000
processor node type=myC40
```

You should consider the following points before designing a mixed-processor application:

- Put the C6000 Diamond installation folder before the C4x Diamond folder in the search PATH to make sure you get the C6000 version of the config command. You can check which version you are getting by giving the command config with no arguments. The C4x version is V3.x.
- Currently C6000 nodes do not support the run-time debug protocol used by C4x nodes. The -A configurer option must therefore be used to suppress the debug information. If you forget the -A switch, the server will be unable to load the application. However, using -A means that you cannot do source-level debugging of C4x nodes in a mixed network.
- Virtual channels and host I/O traffic cannot pass between C4x and C6000 nodes.

- Non-root C4x nodes may only access the server (by using `printf`, for example) if the root processor and all intervening processors are C4x nodes. An equivalent limitation exists for C6000 nodes.

Reserved Hardware Resources

Diamond reserves or requires special treatment of the following resources local to each TIM module:

Resource	Usage
Part of internal memory	The kernel usually occupies part of internal memory. Look at the listing file generated by config option -l to find out which parts of memory are available. Enabling the cache will move the kernel's code into external memory.
EDMA and DMA	Diamond dynamically assigns EDMA channels EDMA4..EDMA7 to operations using Sundance peripherals (comports and SDBs). If a new concurrent comport I/O operation is started and no suitable EDMA channel is free, one comport interrupt will be generated per word of data transferred until one of the necessary EDMA channel is released (whereupon that channel will at once be claimed for any active comport I/O operation that does not have one). Any number of concurrent comport I/Os can be handled in this way. **Warning** This dynamic assignment of EDMA channels to I/O operations means that if you want to control an EDMA channel directly, you must explicitly claim it from the kernel, using the EDMA functions. Processors with the older DMA channels use a similar technique; see DMA functions.
DMA global count reload register B	The C6201 and C6701 kernels' comport device drivers reserve this register. User code must not modify it.
INT4, INT5, INT6	These interrupts are connected to the module's FPGA and are used by Diamond to control comport I/O. If you need to use one of these interrupts directly (e.g., for SDB I/O), you must explicitly claim that interrupt line from the kernel by first calling one of the EXT_INT functions.
INT7	INT7 is also connected to the FPGA but this interrupt is permanently reserved by Diamond's comport device driver so that at least one interrupt line is always available for comport I/O.
INT14 (timer 0)	Reserved. The kernel initialises timer 0 to interrupt once every millisecond. These interrupts support task time-slicing and the `<timer.h>` functions. The kernel handles the timer 0 interrupt specially; you cannot attach your own handler to it.

Byte Order

Diamond assumes that the C6000 processors will be used in little-endian mode, that is, increasing memory addresses indicate increasing significance of the bytes within a word.

Warning

There is no support for big-endian operation.

Confidence Testing

This section describes a short procedure that may be followed to check that Diamond has been installed correctly and is working. The procedure deliberately goes step-by-step to produce a simple application. You can also try other examples, which can be found in `<Diamond>target\c6000\Sundance\examples`. The examples have a descriptive README.TXT file, a batch file for building and running, and an equivalent MAKEFILE suitable for processing by a utility such as NMAKE.

Set the current disk drive to the one on which Diamond has been installed. For example, if Diamond has been installed in folder `C:\3L\Diamond\`, do this:

```
D>c:
C>
```

Set the current directory to a convenient folder for doing this test. For example:

```
C>cd \mine
C>
```

NB: Don't use the installation folder for the confidence test, as this would mean that you would not be testing whether the correct search path has been set up.

Check that the server program is available, by typing the following command:

```
C>3L x
```

This should start the Windows Server. If the server's window does not appear, check that your PATH has been set correctly.

Stop the server by clicking on the x at the top right hand corner of the window.

Copy the example files to the current directory. If the installation folder is C:\3L\Diamond\, for example, you should type this:

```
C>copy C:\3L\Diamond\target\c6000\Sundance\examples\example-1\*.*
…
6 File(s) copied
C>
```

Compile the example using the following command:

```
C>3L c hello.c
```

Build the hello task by linking the resulting object file with the necessary parts of the run-time library:

```
C>3L t hello
```

The examples provided with Diamond all assume that you are using the default processor type. This type will have been defined by the Diamond installation procedure. If you have declined to set a default processor type during installation, or if your processor has changed, you should either:

1. Use the command ProcType to make your new processor the default type; or
2. Edit the configuration file **`hello.cfg`** and change the **`TYPE`** attribute of the **`PROCESSOR`** statement to specify the type, which is appropriate for your particular C6000 board.

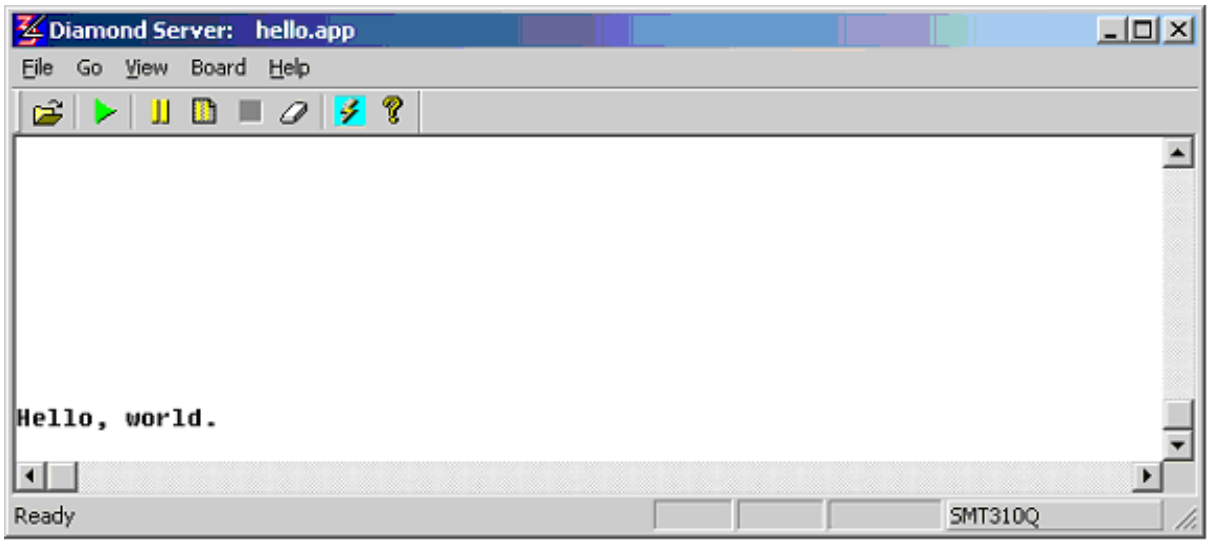
Now generate an executable application, **`hello.app`**, by calling the configurer:

```
C>3L a hello
```

Finally, the program can be run:

```
C>3L x hello
C>
```

The server's window should come up and look similar to this:



The output "`Hello, world`" comes from the **`hello.c`** example program. If it does not appear, we recommend that the installation procedure should be carefully checked and repeated. You should consider, in particular, if one of the following things has gone wrong:

- Has the C6000 board been correctly installed and set up?
- Have the environmental variables described above been correctly set up?
- Is the processor type set correctly in the configuration file? If you have used the type DEFAULT, check using the ProcType utility that you have selected the correct type.

If the "Hello world" message does not appear even after checking all this and repeating the installation procedure, please contact your dealer for further assistance.

Chapter 3. Getting Started

This chapter aims to help you become familiar with Diamond and its terminology. If you have used 3L software on another processor you will already be familiar with the ideas on which the C6000 version is based. If not, don't worry; the strength of Diamond is that its concepts are quite simple. They are explained in outline here, and again in more detail in the chapters which follow.

Overview

The way you build and run applications using Diamond differs substantially from the more traditional techniques used in other environments, particularly Code Composer Studio (CCS). CCS has been designed to produce applications for single processor systems; multiprocessor systems are seen as several separate applications that just happen to be executed at the same time. When Diamond was originally designed almost twenty years ago, it took the diametrically opposed view and considered multiprocessor systems as an integrated whole. This, and the philosophy of keeping things as simple as possible, has led to a system that is both easy to learn and extremely powerful in generating efficient multiprocessor applications.

Note

Diamond does not attempt to decompose your code into pieces that will execute on the different processors in your system; you must do this yourself, based on your unique knowledge of the requirements of your application. Many attempts have been made to achieve automatic allocation of code to processors, and none has come anywhere close to the performance most users require.

Diamond uses a three-stage approach to building an application:

1. Compile your source files with the Texas Instruments compiler;
2. Use the TI linker, usually several times, to generate a number of separate task files;

Warning

You do not usually need linker command files and, even in the cases when you do use them, you never use the MEMORY or SECTIONS [a] directives.

[a] There is one use of SECTIONS in Diamond to work round a problem introduced by changes in the Texas Instruments linker, but this should be invisible to most users.

3. Use the configurer to combine your task files into a single application file that contains everything needed to get your application running on a network of DSPs. Often you do not need to worry about allocating memory to your application; this is done automatically for you by the configurer, although you can exercise more control when necessary.

Once a Diamond application has been constructed, it is loaded into your DSPs and executed by a server program running on the PC.

Warning

A Diamond application file is not a COFF file and cannot be loaded using Code Composer; COFF files do not have the constructs necessary to support the sophisticated multiprocessor loading techniques required by Diamond.

To get a feel for how you work with Diamond you should follow the confidence test described in the previous chapter.

Abstract Model

Diamond's model of parallel processing is based on the idea of communicating sequential processes, as described by C. A. R. Hoare[Hoare]. In this model, a computing system is a collection of concurrently active sequential processes that can only communicate with each other over channels.

> A **channel** can transfer messages from one process to exactly one other process. A channel can only carry messages in one direction: if communication in both directions between two processes is required, two channels must be used. Both the sender and receiver must agree on the size of message being transmitted. Channels are blocking: a sending or receiving thread will wait until the thread at the other end of the channel attempts to communicate.

Each process can have any number of input and output channels, but note that the channels in this abstract model are fixed; new channels cannot be created during the operation of a system.

For example, a disk copy command built into a computer's operating system could be described as three concurrently executing processes: two floppy disk controller processes and one process doing the copying.

This example shows an important property of channel communications: they are synchronised. A process wanting to send a message over a channel is always forced to wait until the receiving process reads the message; this is known as blocking communication. [1] In our example, this means that even if at some time the output floppy disk can't keep up with the input, the system will still work properly. This is because the copy process will automatically be forced to wait if it tries to send a message before the output disk process is ready to receive it. Sometimes it is useful to allow a sending process to run ahead of a receiving one; in such cases an explicit buffering process must be added to the system.

Note that because a process in this model is connected to the outside world only by its channels, the actual implementation of any individual process is not important. A process could be a bit of hardware or a software module; in particular it may also be another complex system, itself consisting of a number of communicating processes.

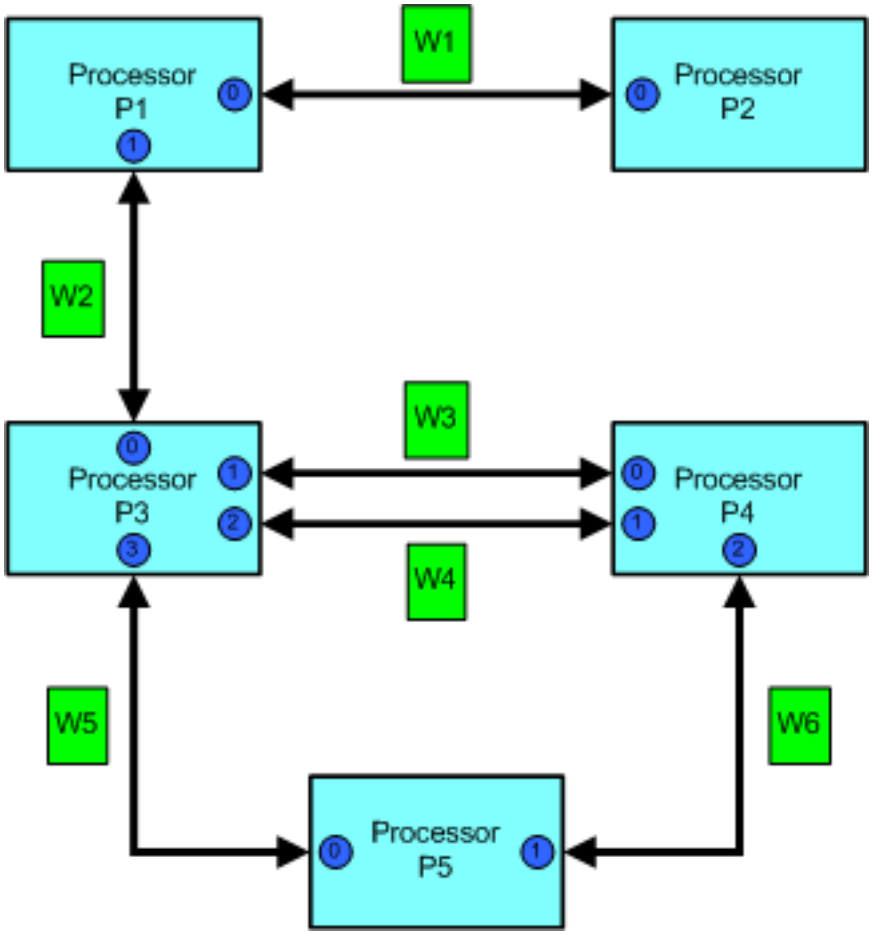
The Diamond Model

The Diamond model for parallel processing closely follows the abstract model described above, but with some additions and relaxations.

Processors, Wires, and Links

The hardware on which a Diamond application runs is described as a network of processors, each with a number of links connected in pairs by wires.

[1] The first reaction of many readers used to queueing models of communication is to think something is missing, however the blocking nature of channels is actually an important simplification. The effect of non-blocking communication is achieved in Diamond by making the application multi-threaded.



Each wire is a two-way communication path between exactly two processors. In practice, different manufacturers provide a variety of ways to connect processors: dedicated point-to-point hardware, shared memory, bus structures, and many more. [2] Diamond hides the details of this from you and behaves as though real wires exist, giving you access to them through a sequence of links numbered from 0. In the diagram above, processor P1 has two links: link 0 connects to processor P2 using wire W1, and link 1 connects to processor P3 using wire W2. As long as you can connect the processors using supported devices, Diamond can create applications for systems of any complexity and any intermixture of hardware types.

The Root Processor

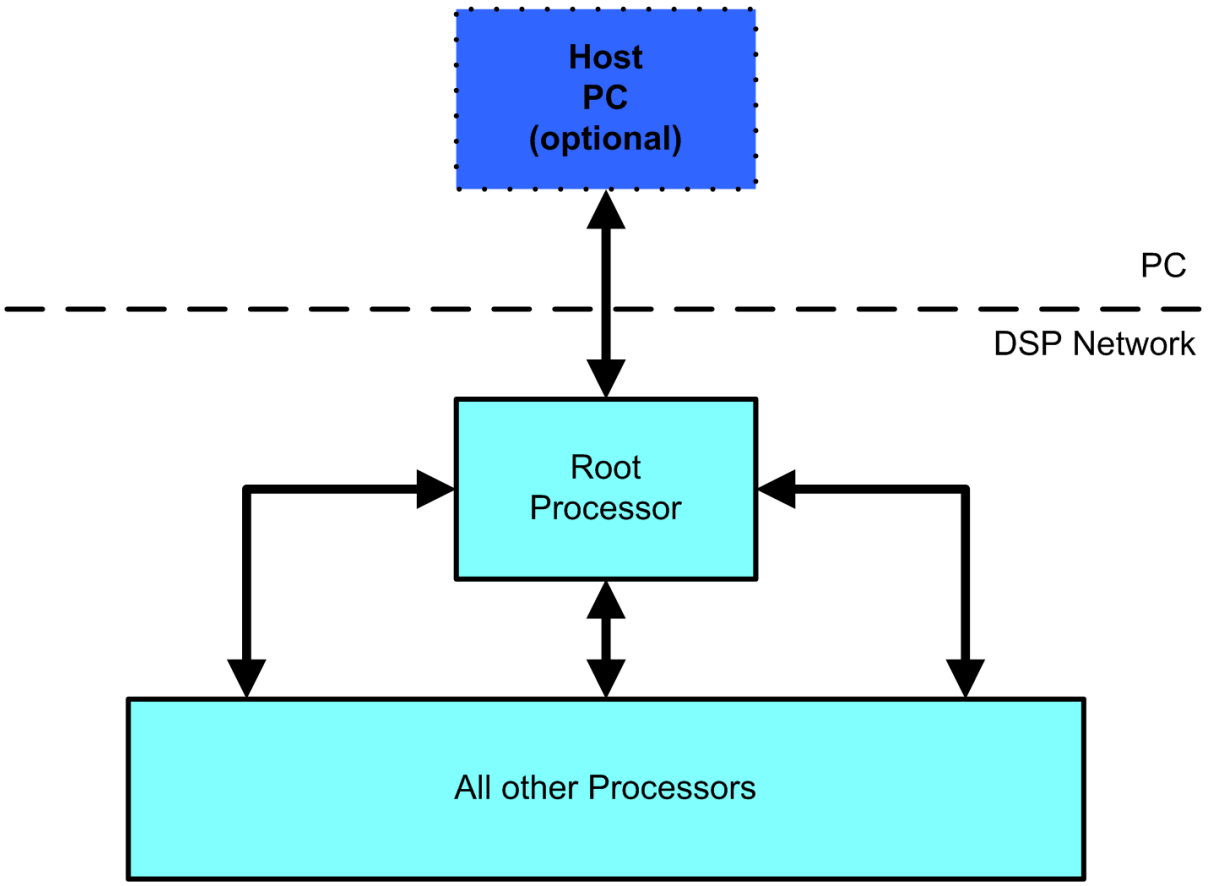
An important feature of all processor networks supported by Diamond is that they are connected. Every processor in the network can be reached by following wires through other processors; no processor is isolated. Because of this, it is possible to identify each participating processor uniquely by choosing one to start from and giving a sequence of links to follow. In example above, starting with P1, we can locate P5 by following link 1 and then link 3.

The starting point in every Diamond application is a processor called **`root`**. The root acts as the base of the network and a reference for locating all other processors.

The Host

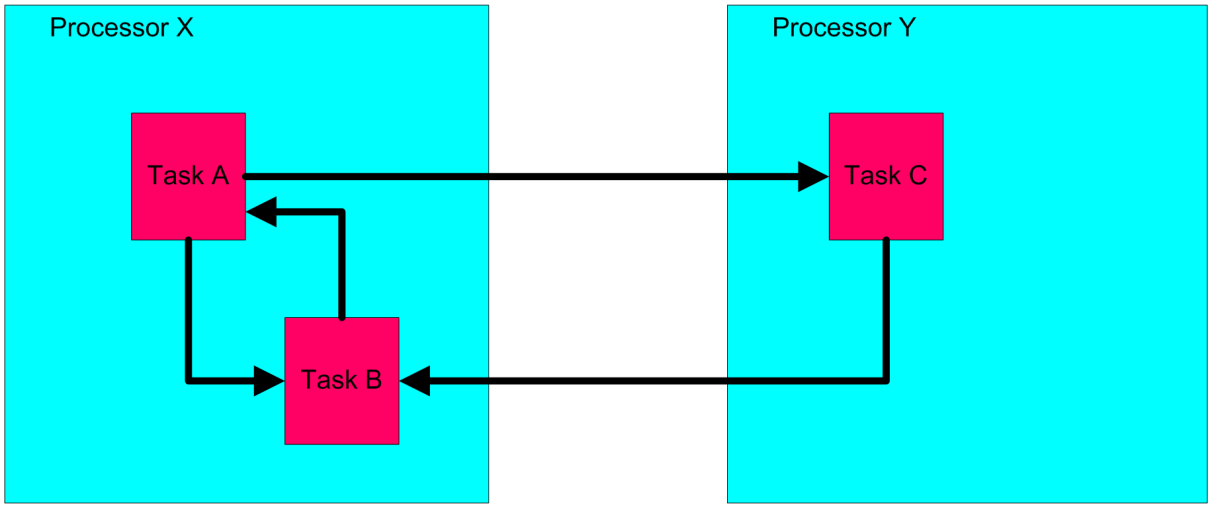
Diamond applications usually run on DSP boards plugged into a Host PC, but this is not a requirement. When present, the host is responsible for loading the application and communicating with it. While it is possible to write your own host code to do this (see here), it is simpler and more usual to leave these details to the host program that Diamond provides, the *server*.

[2] JTAG connections cannot be used as Diamond links



Tasks

A complete application is a collection of one or more concurrently executing tasks connected by channels. Each task is a separate C program, with its own main function and its own regions of memory for code and data. Diamond tasks must obey the register conventions defined for the TI C compiler. The code of a task is a single task image file generated by the linker. Diamond tasks are relocatable, that is, they do not contain fixed memory addresses. When you link a task you do not usually need a linker command file; memory allocation is done at a later stage. You explicitly place each of your tasks on the appropriate processors.



You may have as many tasks on each processor as you wish; you are limited only by the available memory. You may even have several copies of a task. Each processor will be loaded with only those tasks you have specified;

placing a task on one processor does not cause it to be loaded on any other.

Diamond recognises two types of tasks: full tasks and stand-alone tasks.

Full Tasks

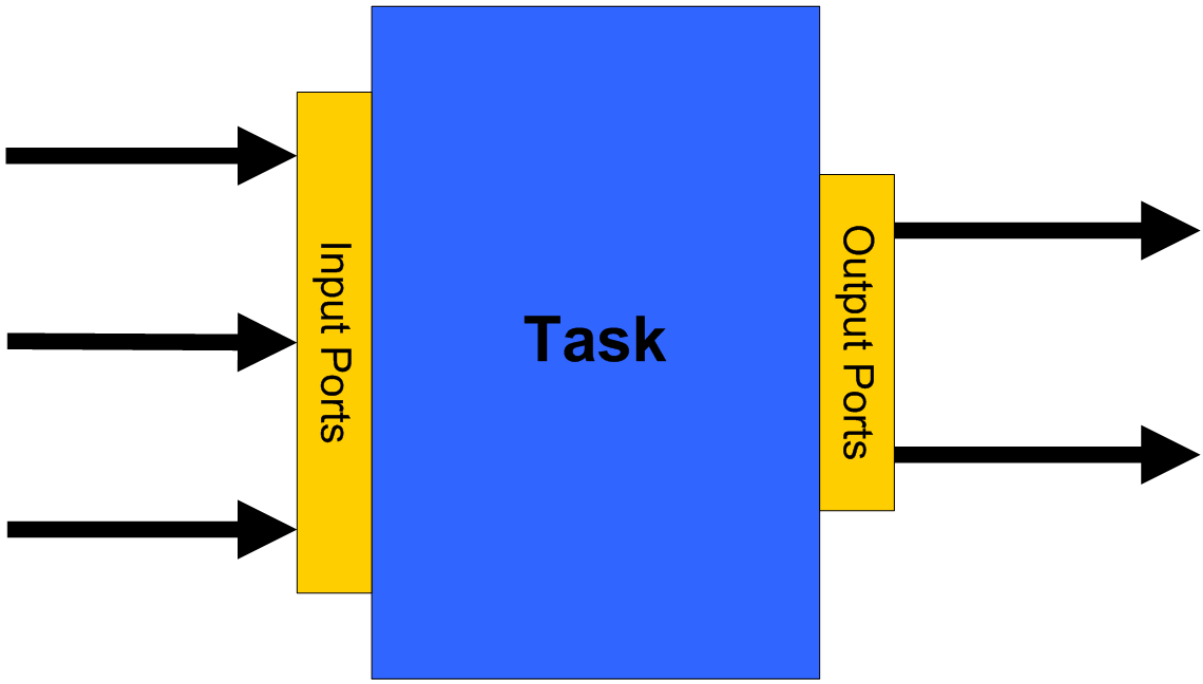
A full task is one that has been linked against the standard run-time library (rtl.lib) with the command **`3L t`**. This gives the task access to the host PC via the server. Full tasks can communicate with the server using the standard C I/O functions, such as `printf` and `scanf`.

Stand-Alone Tasks

Stand-alone tasks are linked against the stand-alone library (sartl.lib) with the command **`3L ta`**. This library omits the host communication functions. In particular, tasks linked against the stand-alone library do not have `stdin`, `stdout`, and `stderr`, and do not retrieve any command-line parameters on starting.

Ports

Each task has a vector of "**`input ports`**" and a vector of "**`output ports`**" that are used to connect tasks together. A task is like a software "black box", communicating with the outside only via its ports [3], as shown below.



You join tasks by connecting output ports to input ports using channels, and this collection of tasks is combined into a single application file by a utility called the *configurer*.

Arguments to main

The standard declaration for a task's main function is:

```
int main(int argc, char *argv[], char *envp[],
         CHAN * in_ports[], int ins,
         CHAN *out_ports[], int outs)
```

[3] A task can include code to deal explicitly with external devices. If you do this, you must place the task on the processor with the devices in question.

argc	number of command-line arguments. This is always zero for stand-alone tasks.
argv	command-line arguments
envp	always null
in_ports	vector of input ports
ins	number of entries in the vector in_ports
out_ports	vector of output ports
outs	number of entries in the vector out_ports

For example, a simple task might accept a stream of values on an input port representing characters, convert each character to upper case, and output the resulting stream of characters on an output port. The code for this is shown below; if you want to experiment you can find the source text in the file: `target\c6000\Sundance\examples\example-2\upc.c`.

```
#include <chan.h>
#include <ctype.h>
#include <stdio.h>                // for EOF

main(int argc, char *argv[], char *envp[],
     CHAN * in_ports[], int  ins,
     CHAN *out_ports[], int outs)
{
   int c;
   for (;;) {
      chan_in_word(&c, in_ports[0]);
      if (c == EOF) break;      // terminate task
      chan_out_word(toupper(c), out_ports[0]);
   }
}
```

Ports and Channels

Tasks can be treated as atomic building blocks for parallel systems. They can be joined together, rather like electronic components, by connecting their ports with channels. The connection may be implemented in memory or across an interprocessor link, either directly or through software routing. Channels that connect tasks on the same processor are known as internal channels; those connecting tasks on different processors are called external channels.

The vectors of input and output ports are passed to a task as arguments of its main function. Each port is of type "pointer to channel" (`CHAN *`). The Diamond run-time library provides a number of functions to send and receive messages over channels.

The configurer can be told to create a number of connections. Each connection is one input channel connected to one output channel. The configurer binds these channels to task ports to create a path from a given output port of one task to a given input port of another. You can access these channels from the task either by indexing into the port vectors passed as arguments to main, or by using the name of the connection to locate the channel directly. See `INPUT_PORT` and `OUTPUT_PORT`.

Multi-Processor Networks

Even though each processor can support any number of tasks, limited only by the available memory, communicating tasks need not be on the same processor. In fact, the tasks of an application may be spread over a large network of processors.

Each channel communication from one processor to another is carried by a link, a physical connection or wire, between two processors. It may be implemented in a variety of ways, but it is capable of supporting communications in both directions.

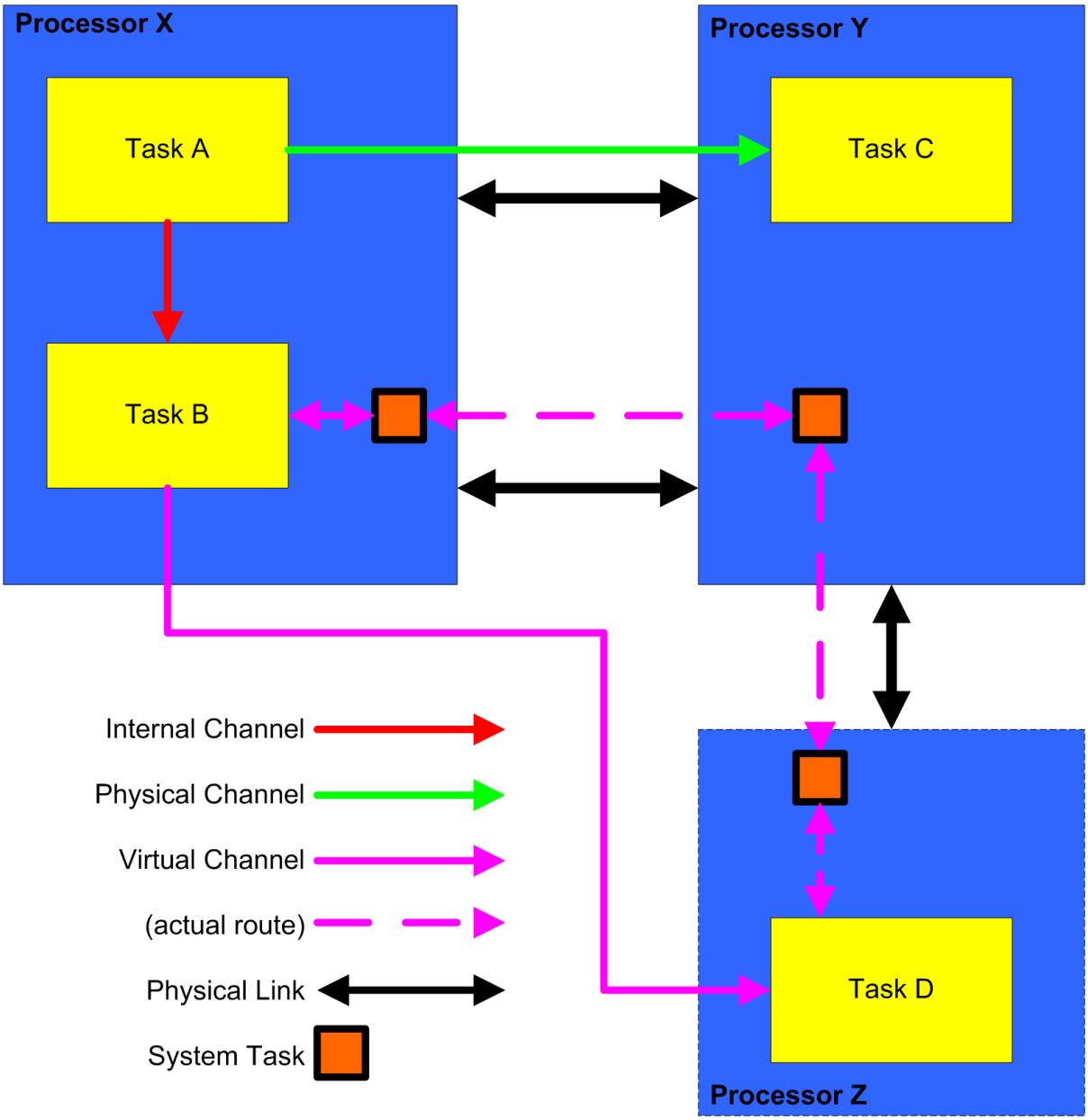
If a channel connects tasks that are on different processors, the messages on that channel are routed through a link between the two processors. Each link can support only two physical channels, one in each direction. You can choose to use these two physical channels explicitly or allow Diamond to manage them for you to give any number of slower virtual channels. Your network can only be built if there are enough links between two processors to support all the required channels.

Any task may communicate with any other task, regardless of where it is in the network. Messages can be routed via the inter-processor links over virtual channels. In addition, all tasks van be given access to C standard I/O.

Virtual and Physical Channels

By default, Diamond makes all external channels virtual, that is, they automatically carry messages between distant processors via intermediate network nodes. Since multiple virtual channels can share the same physical link, any number of virtual channels may connect tasks on different processors. Transmission over virtual channels is guaranteed to be deadlock free.

However, virtual channels are slower than the underlying physical links. Diamond therefore also provides physical channels, which map directly to the hardware. Their advantage is speed: their disadvantage is that the number of physical links, and the paths they take, are both restricted by the available hardware.



When the throughput of particular channels is critical to application performance, you may try making the connection direct and substituting physical channels for virtual ones. Experimenting is easy because the same message-passing functions are used in both cases: no code changes are required.

Simultaneous Input

Sometimes, a task needing to read from two of more channels cannot know which channel will next be ready to transfer a message. The task cannot simply read from each channel in turn, because that would suspend the task until the chosen channel became ready; other channels ready to transfer messages would have to wait.

Diamond solves this problem with the `alt.h` group of library functions. These functions allow a program to wait until any one of a selected group of virtual channels becomes ready to communicate. The channel that becomes ready first is identified to the calling program, which can then go on to read its message using one of the same channel I/O functions used to send messages between tasks.

Parallel Execution Threads

Diamond supports multi-threaded tasks. Tasks dynamically create new execution threads by passing a pointer to a function and an amount of workspace (stack) to a library function. The new execution thread then starts executing the code of the function concurrently with the thread that created it. The new thread runs in the same context as its creator; they share their static, extern and heap memory areas. The only private storage available to the new thread is its workspace. The parent thread has no direct control over its offspring, which continues to execute until it terminates itself by returning from the function that was invoked, or by calling a special library function. This is similar to the execution threads of Win32 and some flavours of Unix. Each thread has its own stack but shares the rest of its data with all the other threads in the same task.

Threads are most commonly created during the initialisation phase of a task; frequent creation and termination of threads generally indicates a poor application design.

Semaphore functions in the run-time library can be used to prevent threads that share data from interfering with each other. Alternatively, internal channels declared as program variables can be used to synchronise the threads' operations and transmit data between them by passing messages. Diamond provides a `CHAN` data type that can be used to declare channel variables.

Of course, like any other software construct, threads may be used in contexts other than those in which they are formally necessary. Indeed, many problems in simulation, real-time control and other areas map well onto a multi-threaded algorithm, although they do not strictly require to be executed in this way.

Once a task starts, its main function behaves just like a thread; in particular, it may be stopped using `thread_stop`.

Configuring an Application

Once an application has been designed and written as a collection of communicating tasks, how is it loaded into a physical network of processors?

First, each individual task is built by compiling all its source files with the C compiler and using the linker to combine the resulting object files with the necessary modules from the run-time library. Repeating this for every task in your application results in a number of task image files.

Now the task image files must be combined to form a single executable application file. The program that does this is called the *configurer*. A user-supplied textual *configuration file* drives the configurer and specifies:

- Your hardware structure:
  - available processors
  - links connecting them
- Your software structure:
  - tasks to be included
  - channel connections between them
- How to map the software onto the hardware.

The configurer allocates memory for the tasks and combines them into a single application file that can be loaded into the specified hardware network and executed using the Diamond command, **`3L x`**. The configurer is also responsible for determining which system tasks need to be loaded.

Note

To change the way in which tasks are connected together or the processors on which they are to run, it is not necessary either to change your source code or even to recompile or re-link the tasks themselves. This means, for example, that it is possible to develop an application while running all the tasks on one processor, and then reconfigure it, without any other change, to run on a network. Physical channels may be transparently substituted for virtual ones in a similar way.

The Structure of an Application

So far, we have seen a Diamond application as a network of tasks spread over a number of processors, possibly

with more than one task per processor. These tasks communicate only through channels. If a task wishes to communicate with a task on another processor, messages will be transmitted to that processor across the inter-processor links.

Typically, the network is controlled by another processor, which we call the host. The host communicates directly with only one processor in the network, which we call the root.

In fact, the application also includes a number of other components. We shall look at these one by one.

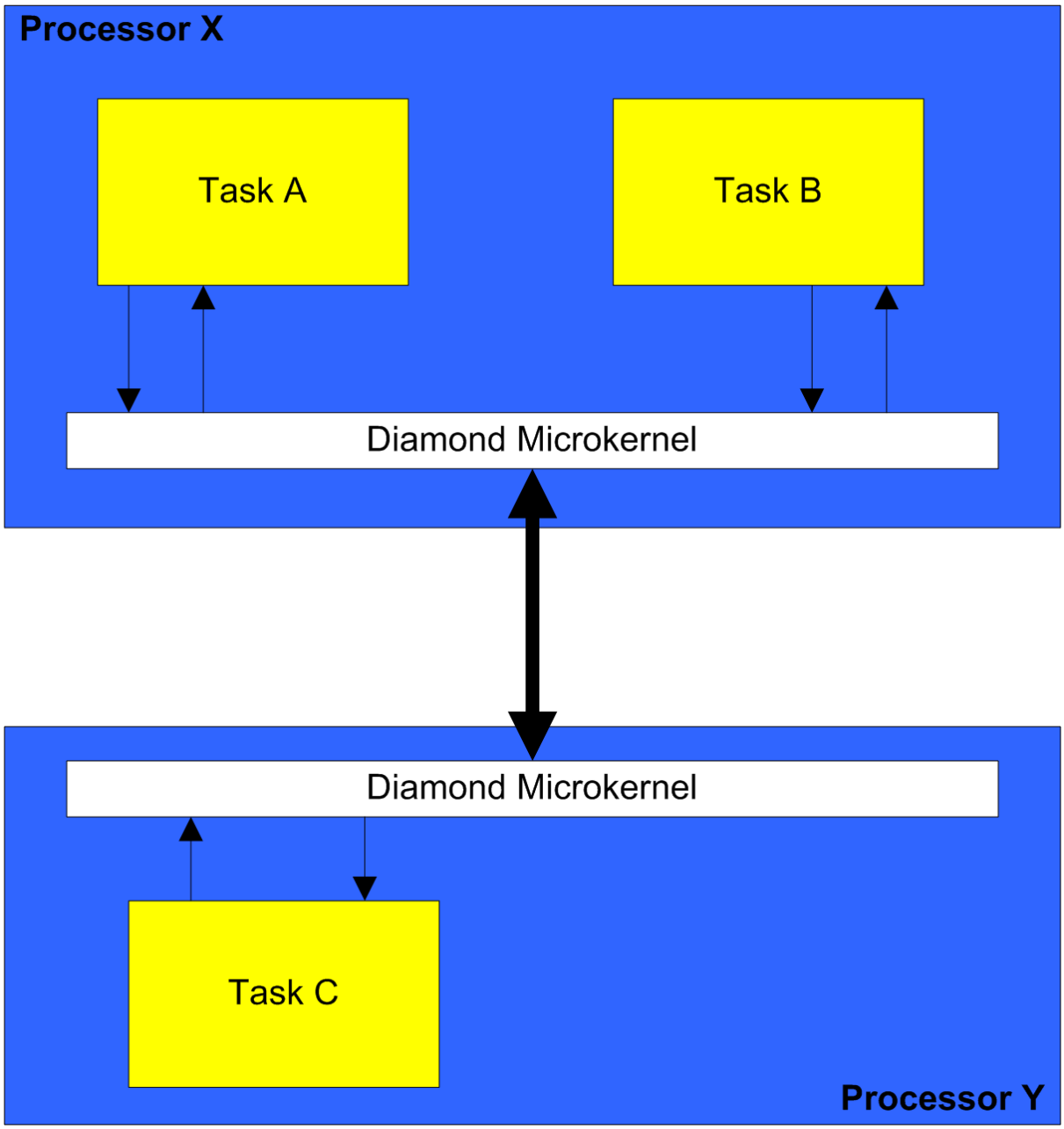
The Microkernel

The configurer automatically places an appropriate microkernel on every processor. Among the jobs the kernel performs are the following:

- scheduling all the threads running on the processor;
- managing the interprocessor links and channels;
- controlling the timer;
- providing the primitives for implementing semaphores and events;
- handling interrupts.

The kernel is a passive component in the system; it only consumes processor cycles when asked to do something, either as the result of an interrupt or an explicit program request (for example, a link or semaphore operation). TIMER0 interrupts are used by the kernel to manage its internal clock (`timer_now`, `timer_delay`, and `timer_wait`), and for timeslicing. Timer management takes approximately 25 cycles every millisecond. If you only have a single thread on a processor, there will be no timeslicing. By setting CLOCK=0 for the processor, kernel clock interrupts and timeslicing will be stopped; all processor cycles will be available to your code, although this extreme step is rarely beneficial.

The kernel is very efficient; its performance does not depend on the number of threads in the system.



Every task is passed a handle to the kernel, visible as the variable `_kernel`. Usually you do not need to be aware of this, but certain functions require it as a parameter (the external interrupt handling functions, for example).

Scheduler

Many tasks can be run on one processor, and tasks themselves are made up of one or more concurrently-executing threads. The microkernel arranges for the available processor time to be shared amongst these various threads using a process known as *scheduling*. Each thread has an associated priority, a number in the range 0–7 [4] , that determines its importance with respect to other threads executing on the same processor; the smaller the number, the higher the priority of the thread. Threads at the highest level of priority (priority 0) are referred to as *urgent threads*.

[4] It is extremely rare for an application to need more than three priorities (often, 0, 1, and 2). Profligate use of multiple priorities can reduce the efficiency of an application.

An executing thread can be suspended (descheduled) and another thread resumed when one of a number of events occurs:

- The thread explicitly requests to be descheduled by calling `thread_deschedule`;
- The thread has to wait for something (a channel communication, a semaphore, an event, or a timer event);
- The thread is pre-empted by a more important (i.e., higher priority) thread that has become ready to execute;
- The thread is timesliced, descheduled because it has been running for some time (its time slice) without interruption. The time slice is computed from the thread's priority: priority*100ms.

Urgent threads are of the highest priority, priority 0, and so can never be pre-empted, even by another urgent thread. Similarly, urgent threads are considered to have an infinite time slice and will execute forever without being descheduled. Lower priority threads, and even other urgent threads, will be "locked out" entirely. Careful use of urgent threads can dramatically improve the performance of an application; injudicious use can equally dramatically degrade performance. For this reason, urgent threads should be used with care. A general rule of thumb is that threads whose primary purpose is communication are good candidates for being urgent. A processor that has one urgent thread (task), has disabled the clock, and does not use any Diamond kernel services, will spend all of its cycles executing the thread.

Once a thread is descheduled, the highest-priority thread that is ready to execute will run [5] . Within any priority level except the urgent level, the microkernel will divide the time amongst the threads on a round-robin basis (first come, first served).

Switching from one thread to another because of an interrupt (including a time-slice) is slightly more expensive than switching for any other reason.

Universal Packet Router (UPR)

The universal packet router is a module that is automatically included in a multiprocessor application when tasks on different processors wish to communicate but there is no direct physical link between the processors.

Global File Services (GFS)

The global file services module is used to give non-root processors access to host file I/O services (such as printf). GFS uses UPR to manage communication between non-root processors and the host.

Virtual Channel Router (VCR)

The virtual channel router allows tasks on processors that are not directly connected by links to communicate across channels. It also can multiplex several such virtual channels across a single physical link. VCR makes use of the services offered by UPR.

The Server

The server is a Windows program that runs on the host and has two important functions:

1. To load the application into the C6000 network;
2. To perform I/O and other operations on the host on the network's behalf.

Because of the second function, the server usually runs on the host all the time that a C6000 application is running. The details of server operations are normally invisible; you simply call C I/O functions such as `printf`. The run-time library converts these calls into appropriate instructions to the server.

Stand-Alone Applications

[5] This means that lower-priority threads will only ever execute when all threads of a higher priority are waiting (for example, for a semaphore or a link transfer).

It is possible to develop Diamond applications that do not use the server. The most common example would be an application that is held in ROM and is loaded automatically when the processors start up. Another example would be an application that is loaded from, and talks directly to, user-written code on the host PC. Such applications have complete control over communications with the host PC, if any.

When building a stand-alone application remember that:

- all tasks must be linked against the stand-alone library;
- you must run the configurer with the **-A** switch.

Performance

Diamond has been designed to allow you to write high-performance applications with the minimum of fuss. The microkernel ensures that the available CPU cycles are spent wisely. To that end, the system is driven by interrupts. When a Diamond thread is waiting for something (communication with the host or another processor, for example), the kernel puts that thread to sleep and passes control to the next thread that is ready to run. A sleeping thread uses no CPU cycles. The thread will be woken up and made ready to run when the event that it has been waiting for occurs.

Context switching performance

Changing from one thread to another is known as context switching, and Diamond is very efficient at doing this. As an example, when using internal memory, context switches (`thread_deschedule`) have been measured as follows:

TIM	Processor	Clock speed	Context switch time
SMT374	6713	225MHz	~470ns
SMT361	6415	400MHz	~250ns
SMT395	6416T	1GHz	~100ns

Chapter 4. Sequential Programs

This chapter shows you how to use Diamond to write conventional sequential programs to run on the C6000. You should be familiar with the contents of this chapter before you progress to the later chapters explaining parallel programming.

The instructions in this chapter assume that the software has already been installed.

Overview

The process of converting one or more source files into an application ready for running takes three steps:

1. Compiling source files into object files;
2. Linking sets of object files into tasks; and
3. Configuring one or more tasks into an executable application.

Compiling

Each source program file is compiled into an object file containing C6000 instructions. You can compile a source file by a command of the following form:

```
► 3L c source-file
```

The **3L c** command invokes the Texas Instruments C compiler with certain required option switches.

A source-file is the name of a C source file, and the filename extension "**.c**" must be given in the command. So, to compile the file **hello.c**, you would give the command:

```
► 3L c hello.c
```

If the source file contains no errors, an object file **hello.obj** is produced. If the compiler detects errors in the source program, it writes diagnostic messages to the screen. For a discussion of the format of these error messages and how to make use of them, please see section 2.8 of the Optimizing C Compiler.

Compiler Option Switches

Apart from the switches that the 3L c command uses automatically, the compiler has a number of others, which are described in the Optimizing C Compiler[C Compiler]. If you need to use the switches, you can specify them in the command, after the source file name, like this:

```
► 3L c hello.c -dDEBUG -k
```

This will compile the program **hello.c**, with the macro DEBUG defined. The output assembler file will be retained at the end of the compilation, instead of being deleted after the assembler has finished.

There are compiler switches that can invoke the linker once the compilation has been completed, but we recommend these should not generally be used when Diamond applications are being built. See below: linking.

Calling the Compiler Directly

Sometimes you may wish to invoke the compiler directly rather than via the **3L c** command. In those cases, you must use the following required switches normally supplied by **3L c**:

Switch	Function
-c	Do not run the linker
-I"<Diamond>bin\c6000\Sundance\include"	Get header files from the Diamond folder. You must replace <Diamond> with the actual path of the installation folder. This is usually c:\3L\Diamond\

For a discussion of the compiler switches "-c" and "-I", and a discussion of the various levels of optimisation available, see the Optimizing C Compiler[C Compiler].

FAR Data

As detailed in the TMS320C6000 Optimising C Compiler Manual, if a task's static data is larger than 64KB, the task must be compiled using the large model. In addition, as detailed in section 7 of that manual, the compiler does not initialise any unassigned static data to zero. Diamond will initialise near static data (in the .bss section), but does not initialise far static data. Such data can be initialised either as detailed in the C manual[C Compiler] or by combining the .far and .bss sections at link time. See the TI Assembler tools manual for examples of how to do this.

Linking

Once a program has been compiled into one or more object files, the program must be linked with any external functions it requires before it can be run, including functions like `printf` from the run-time library. The Texas Instruments linker does this, and the result is called a task image file.

Diamond's task image files are relocatable, that is, the various code and data areas they contain have not yet been allocated to particular memory addresses in the target processor. This will be done during a subsequent configuration step. For this reason you must not supply linker command files with the MEMORY and SECTIONS that would be necessary when using TI's tools alone.

Here we discuss the most usual linker operations; more about the linker can be found in the Assembly Language Tools.[Assembler]

You can link together any number of object files to form a task image with a command of the following form:

```
3L t object-file object-file …
For example:   ►  3L t hello
```

`3L t` is a command that invokes the linker in the most common way. Each object-file should be the name of an object file produced by the compiler; you need not supply the ".obj" extension. If the object files are specified without extensions, the linker assumes the extension ".obj".

Warning

The name of the first argument will *always* be used to construct the name of the task image file by replacing any extension with ".tsk".

So in the example above, hello.obj will be linked with the necessary modules from the library, and a task image file hello.tsk will be created. Note that any linker switch to specify a different task image name (-o) will be ignored; to change the name you will need to call the linker directly and not use this command.

If you specify more than one object file, the **`3L t`** command constructs a task file name from the first parameter. For example, if there are two C source files, main.c and fns.c, the following commands will compile them and link them together to produce main.tsk.

```
►  3L c main.c
```

```
► 3L c fns.c
► 3L t main fns
```

It is possible to include linker option switches in a **3L t** command. Assembly Language Tools discusses the linker's switches in detail. For example, the following command will place a linker map file in main.lis:

```
► 3L t main fns -m main.lis
```

Using Linker Command Files

The following three sections deal with more advanced uses of the linker. If you want to build only fairly simple programs for now, you can skip to the section on running programs.

Sometimes you may have more object files to link than is convenient or possible to include on the connad line. In this case, you will have to use a linker command file. In its simplest form, a linker command file is a text file containing a list of object file names, all of which are to be included in the executable file. For example:

```
► 3L t myprog.ind
```

This will cause the linker to find the file myprog.ind, and link together all the object files specified in it, include the necessary modules from the run-time library and generate a task image file called myprog.tsk.

The contents of myprog.ind might be as follows:

```
main
fns
-i grafpack.ind
```

This will link the object files main.obj and fns.obj, together with all the object files specified in the command file grafpack.ind.

Further details about the linker, and linker command files, may be found in Assembly Language Tools. Note that Diamond users should never [1] use the linker's MEMORY or SECTIONS directives.

Libraries

It is often convenient to be able to treat a group of object files as a single unit. For example, the Diamond run-time library consists of many separate object files, but is supplied as a single file containing all of them. As we have seen above, the linker will search libraries for the necessary modules to complete the program being linked.

Using a library has several advantages over using a linker command file:

- The linker selects from the library file only those modules that are actually referenced elsewhere in the program; the others are not included in the task image file.
- Copying a single file to another place is simpler than copying many component object files and making sure that the corresponding linker command file is kept up to date with changes in folder and file names.
- Opening just one library file is faster than opening a linker command file and several object files.

Library files are created and maintained by the archiver. Details of this program may be found in Assembly Language Tools[Assembler]. The example below shows a graphics library, grafpak.lib, being built from a core graphics module, core.obj, and two device driver modules, tek.obj and hp.obj:

[1] There is one exception where a SECTIONS directive has been used to work round difficulties caused by changes Texas Instruments made in the linker, but this is usually hidden in Diamond commands.

```
► ar6x -a grafpak.lib core.obj tek.obj hp.obj
```

The default extension for the library name is ".lib". The archiver does not assume an extension for the object file names; you must specify ".obj" for each one.

The following example shows how to replace the module hp.obj with a new version:

```
► ar6x -r grafpak hp.obj
```

Note

It is probably safer always to use the –r switch with the archiver to ensure that only the more recent versions of files are kept.

Calling the Linker Directly

Sometimes you may wish to call the linker directly, instead of using the **`3L t`** or **`3L ta`** commands. There are some required options that you must use every time you link Diamond tasks; the easiest way to get them is to use the standard Command.dat files as templates. The following is a batch file that has the same effect as **`3L t`**:

```
echo  -ar -cr                                                  > tmp.cmd ❶ ❷
echo  SECTIONS                                                >> tmp.cmd ❸
echo  {                                                       >> tmp.cmd
echo     .bss:   { *(.bss)   }                                >> tmp.cmd
echo     .cinit: { *(.cinit) }                                >> tmp.cmd
echo     .text:  { *(.text)  }                                >> tmp.cmd
echo  }                                                       >> tmp.cmd
echo  file1                                                   >> tmp.cmd ❹
echo  file2                                                   >> tmp.cmd
echo  file3                                                   >> tmp.cmd
echo  file4                                                   >> tmp.cmd
echo  -l "<Diamond>bin\c6000\Sundance\lib\rtl.lib"            >> tmp.cmd ❺
echo  -l rts6200.lib                                          >> tmp.cmd ❻
echo  -o %1.tsk                                               >> tmp.cmd
lnk6x -qq tmp.cmd
erase     tmp.cmd
```

❶ The -ar switch makes the linker generate a relocatable task image file. We need this, since the configurer will later decide the absolute locations where the sections of the task will be loaded. See section 7.4.1 of Assembly Language Tools[Assembler].

❷ The -cr switch instructs the linker to build a task that follows the conventions for a C6000 C program. It also requests that static variables should be initialised at run time; the Diamond environment requires this. See section 7.16 of Assembly Language Tools.

❸ SECTIONS is there to cope with a change introduced in recent TI code generation tools. Without this SECTIONS incantation, the linker will no longer combine all the necessary subsections of .bss, .cinit, and .text into single areas, and the configurer will subsequently report errors.

❹ The example assumes you are linking four files named `file1.obj`, `file2.obj`, `file3.obj`, and `file4.obj`, but you may link as many or as few object files as you wish. You should, of course, use the names of your own files.

❺ The -l "`bin\c6000\Sundance\lib\rtl.lib`" switch specifies the full Diamond library as a library to be searched by the linker. You can replace "rtl" with "sartl" to build a stand-alone task. You will need to replace with the path to the Diamond installation folder, usually `C:\3L\Diamond\` (note the final '\').

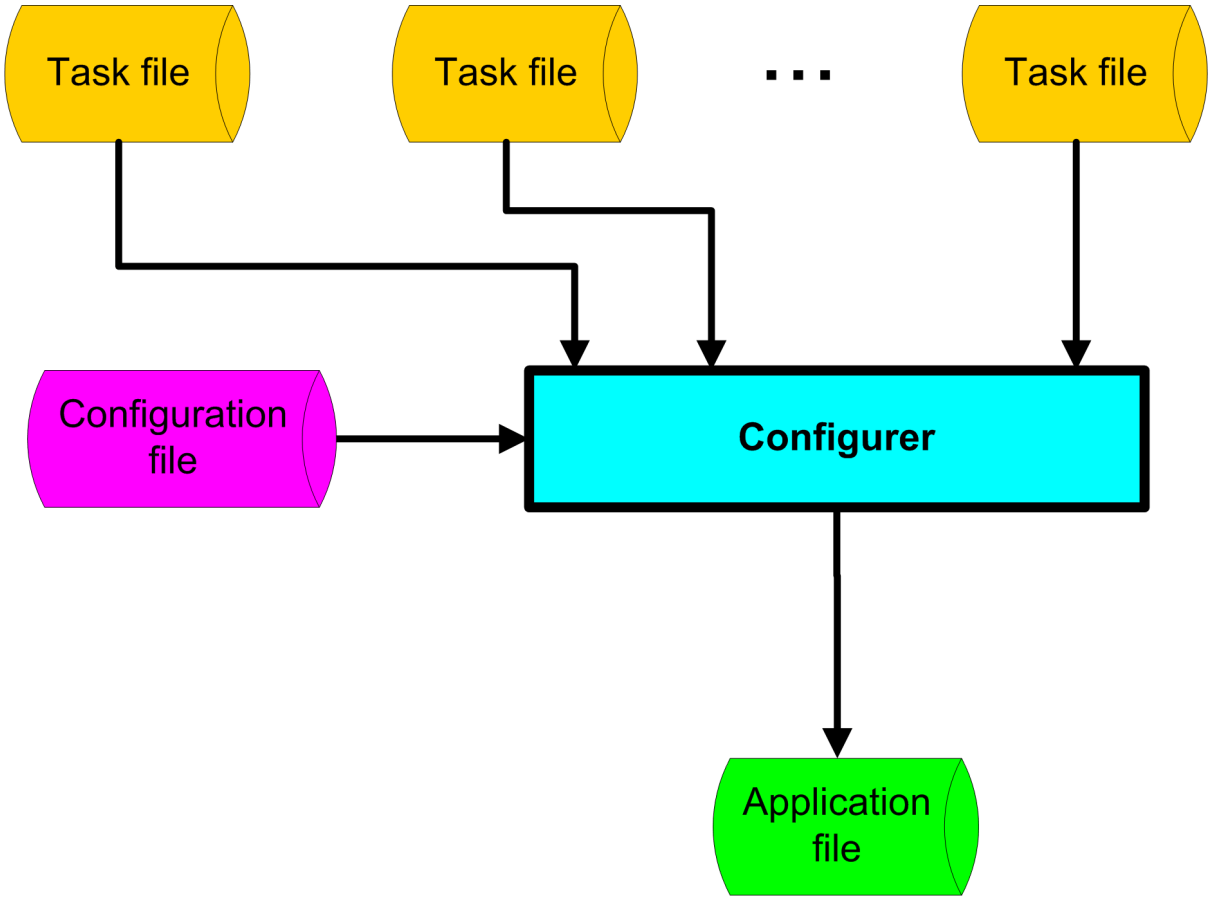
❻ The -l rts6200.lib switch specifies the TI library as a library to be searched by the linker for functions not defined in the Diamond library. It is important that this library is placed *after* the Diamond library.

Locating files

Whenever the linker attempts to open object libraries or command files, it looks by default in your current

directory. You can make the linker search other directories by using one or more **-I pathname** switches.

Configuring



After a task image has been linked, it must be configured. This process decides how the various parts of the program are to be mapped onto your 6000 board's memory, and creates an application file by attaching the 3L microkernel, tasks required for communication, and so on. The program that does this is called the *configurer*. Its full purpose is to bind together a number of Diamond tasks, ready to be loaded into a network of processors, but here we are using it in only a simple way. See: configurer and configuration language.

Calling the Configurer

The first thing you must do before configuring an application is write a configuration file. This is a text file that provides information about the tasks and processors your application will use.

To start with, we shall consider an application that includes only one task running on one processor, so the configuration file we shall need will be simple. For example, the following three-line configuration file (myprog.cfg) could be used to create such an application from myprog.tsk:

```
PROCESSOR root   TYPE=MyBoard ❶
TASK      myprog DATA=? ❷
PLACE     myprog root ❸
```

❶

The **PROCESSOR** statement describes the target hardware we shall be using. It specifies the processor's name, which in this simple case must be root, and its type. The type not only indicates that this is a C6000 processor, but it also identifies which type of C6000 module we are using; MyBoard has been used as a

place-holder for the particular DSP module you want to use. The various modules differ in the way their external memory is laid out, and in the way the certain modules communicate with the host, so it is important to set the type correctly. You can use the special processor type **DEFAULT** to select the type set by the ProcType utility.

❷ The **TASK** statement specifies the name of the task. The configurer will use this to find the task image file. The **DATA=?** clause tells the configurer to allocate a single contiguous area of memory for the application's stack and heap data areas. This area will be the largest area left when the memory needed for everything else (the code and static data, for example) has been allocated. If you omit **DATA=?** the configurer will act in the same way, but will output a warning message. Other ways to allocate memory are discussed in "OPT Attribute".

❸ The final line of the configuration file is a **PLACE** statement, telling the configurer to place the task myprog on processor root.

Now that we have a configuration file we can generate an application file by invoking the configurer with a command of the following form, where the extensions of the names of the configuration file and application file are specified explicitly:

```
3L A  configuration-file  application-file
```

The configurer can also be called with a single file parameter:

```
► 3L A   file
```

This has the same effect as the command:

```
► 3L A  file.cfg  file.app
```

The program myprog described in the example above could be configured with this command:

```
► 3L A myprog.cfg myprog.app
```

The configurer will take the task image file and automatically allocate memory for all the task's components. If you wish to see where the configurer has placed things, you can make the configurer send a detailed listing to the standard output stream by means of the **-L** switch, as follows:

```
► 3L A myprog.cfg myprog.app -L > myprog.lis
```

The configurer allows more explicit control of memory allocation; this is described here, but you are advised to leave memory allocation to the configurer during development.

Running

Applications are usually loaded into the C6000 modules and run by using the server program, WS3L. The server is an ordinary Windows program running on the host computer. After loading the application into the C6000 modules, it usually remains active throughout the application's run. The C run-time library sends instructions to the server whenever it needs to perform host operating system functions such as reading information from the disks, displaying output on the screen and so on. The server sends the results of these operations back to the C6000 modules

You can start the server in four ways:

1. By double-clicking on WS3L.EXE or a shortcut to it;
2. By double-clicking on a file with a .app file type;
3. By dragging a .app file from an explorer window and dropping it into the server’s window; or
4. By giving either of the following types of command from a command prompt:

```
   3L X
or 3L X filename
```

The filename must be the name of an application file produced by the configurer. If you do not supply an extension, the server will add "**.app**" .

All of these commands will bring up the server window. If you have used a **.app** file to start the server, that file will be selected; otherwise you will have to select the application you want from the server's "File" menu.

Before you can get the server to load your selected application into the DSP network and start it running, you must tell the server which DSP boards you are using. Selecting a link interface with the "Board/Select" menu does this. If you haven't selected a board before, the server will automatically select the first one it finds, and if an application file has been selected, the server will start it running.

You can find more information about the server and its options in "The Server".

Command-Line Arguments

The server allows you to specify a command line to be passed to the application it invokes, for use as program arguments. These arguments are passed to every task that has been linked against the full Diamond library; stand-alone tasks are not passed arguments.

By clicking on "View", then "Options", and then selecting the "General" tab, you will bring up a window that allows you to give your command line arguments. The arguments are remembered by the server and will be used each time you run the server until you change them.

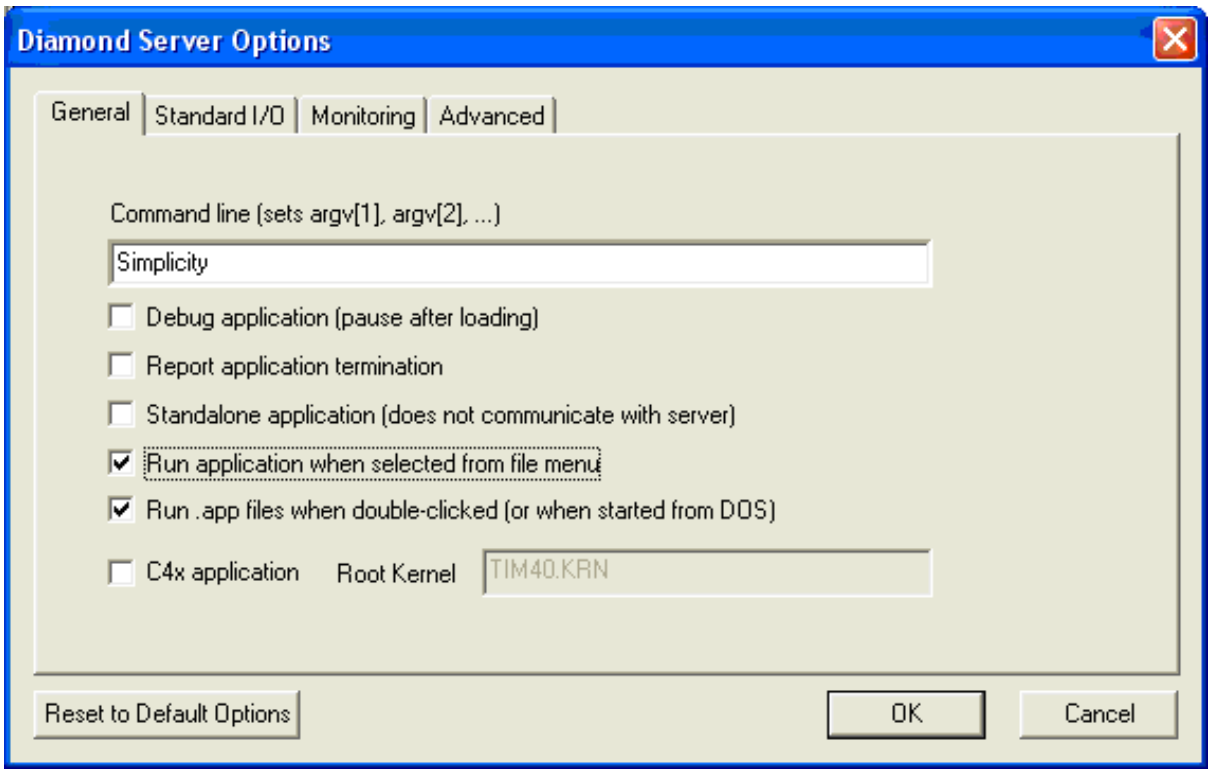
The command line is broken into program arguments, and these are made available to the tasks in an application in the usual C way. When the C `main` function is called, it is passed the following parameters :

```
int main(int argc, char *argv[])
```

argv	is a vector of pointers to the arguments, each of which is a zero-terminated string.
argv[0]	points to the name of the application that is running. It will be NULL if the task is stand-alone.
argv[n]	points to the n^{th} command-line argument;
argc	is the number of arguments, including the program name. It is always at least 1; it will be exactly 1 for stand-alone tasks.
argv[argc]	is always a null pointer.

In fact, the main function has more parameters than these. The others are used for inter-task communications and are discussed in "Inter-Task Communication".

For example, assume we have previously set the following:



The command:

- `3L x myprog`

will start the server with the application file "`myprog.app`" selected. When you run the application, the following values will be passed in:

argc	will be 2.
argv[0]	will point to the string "myprog.app",
argv[1]	will point to the string "simplicity", and
argv[2]	will be a null pointer.

Chapter 5. Parallel Programs

In this chapter we move on from looking at the general features of Diamond to explaining how some of its parallel programming tools are used in practice. The configurer is described here in more detail along with the library functions for sending messages over channels and creating new execution threads.

One User Task

We have already seen how to configure applications consisting of a single user task. It may be useful, however, to take a closer look at what happens when we do this.

The example program we have chosen just converts a stream of characters read from stdin to upper case. Here is the C source file, `upper.c`, for the upper case program:

```
#include <stdio.h>
#include <ctype.h>

main()
{
   int c;
   while ((c = getchar()) != EOF) {
      putchar(toupper(c));
   }
}
```

As we have seen, we could build and run an application from this source file (and a suitable configuration file) with the following commands:

```
► 3L c upper.c
► 3L t upper
► 3L a upper
► 3L x upper
case changer
CASE CHANGER
…
```

Configuration

When building an application, the configurer must be told the details of both the software and the hardware on which it is to run. Supplying it with a suitable configuration file does this. The configuration file must be written by the user, and describes the system to be built: all the physical processors in the system, the wires connecting them, the tasks to be loaded into the system and their logical interconnections.

Earlier, we looked at the configuration file that we needed for our single-task example. It is quite instructive to look more closely at a (strangely) reformatted version of the same file, as shown below.

```
! ❶
! UPPER.CFG
!
processor - ❷
     root -         ! the processor connected to the host ❸
     type=MyBoard   ! change this to your processor type ❹
task upper data=?   ! the single task ❺
place upper -       ! locate the upper-casing task... ❻❼
        root        !   ...on the root
```

❶ To start with, we have a number of comment lines, introduced by a "**!**" character.
❷ The first significant line is a **PROCESSOR** statement: a declaration of the single processor, **root**.

In Diamond terminology, the root processor is the only one that is connected to the host. All communication with the file system on the host must pass through it. Every configuration must include a processor named root. In fact, the host processor could be regarded as part of the configuration as well, but the configurer assumes this; the host processor need not be declared. The root processor, however, must be declared with the name **root**; this informs the configurer that it must be connected to the host in the appropriate way.

❸ You will notice that comments can also occur further along the line, again introduced by a "**!**" character, and that spaces, tab characters and blank lines can be used as desired, to improve readability. The configurer ignores the case of letters: "**a**" and "**A**" are not distinguished.
❹ The type of root is declared as MyBoard (you should change this to refer to the type of DSP module in your system):

```
processor root type=MyBoard
```

The processor type makes the configurer aware of the properties of the processor you will be using, including:

- the processor class (C6000, in this case);
- the sort of module or board supporting the processor;
- how it communicates with other modules and the host (TI-style communication ports);
- the layout and size of available memory areas; and
- the clock speed.

In this edition of Diamond, a type must be specified for every processor.

❺ This line declares the single user task, "upper". It directs the configurer to load the task upper from the task image file "upper.tsk". The TASK statement also specifies the memory allocation strategy to be used for this task. Here, the data=? attribute will result in the stack and heap data areas sharing a single area of memory, this being the largest area left vacant once all the other parts of the task (code, static data, etc) have been loaded. If you omit this clause, the configurer will act in the same way, but will print a warning.
❻ Notice that in the full example this line has been broken; this is indicated by placing a hyphen "-" as the last non-whitespace character before the comment.
❼ Finally, we have a PLACE statement that directs the configurer to place the upper task on the root processor:

```
place upper root
```

In this small configuration file, we have seen two examples of objects (root, upper) that have been declared with configuration language names. Objects like these can have arbitrary names made up of letters, digits and the characters "_" and "$", but are usually given mnemonic names.

Building the Application

Once the configuration file has been written, you can build the application by compiling the program, linking it and then running the configurer. The commands required to build our uppercasing example would be as follows:

```
► 3L c upper.c
► 3L t upper
► 3L a upper
```

Linking

The task image file output by the linker has the same name as the first object file, with the extension ".tsk"; this is the extension expected by the TASK statement in the configuration files.

Configuring

The configurer is invoked by the **3L a** command. One or two filenames must be specified on the command line: first the configuration file, then the name of the application file to be output. For our case-conversion example, the required config command line is:

- `3L a upper.cfg upper.app`

As the file names use the standard extensions ".cfg" and ".app", the command could be simplified to:

- `3L a config upper`

File names for the task image files which make up the application are not supplied on the command line; the configurer derives them automatically by appending ".tsk" to the task identifiers given in the configuration file. In our example, the configurer will search for "`upper.tsk`".

If a task image file is not found in the current directory, the configurer will automatically search all of the directories in the search path, that is, all those specified in the PATH environment variable. PATH can be modified in the usual way, by the path command. This automatic mechanism for specifying task image file names can be overridden by the FILE attribute of the configuration language's TASK statement.

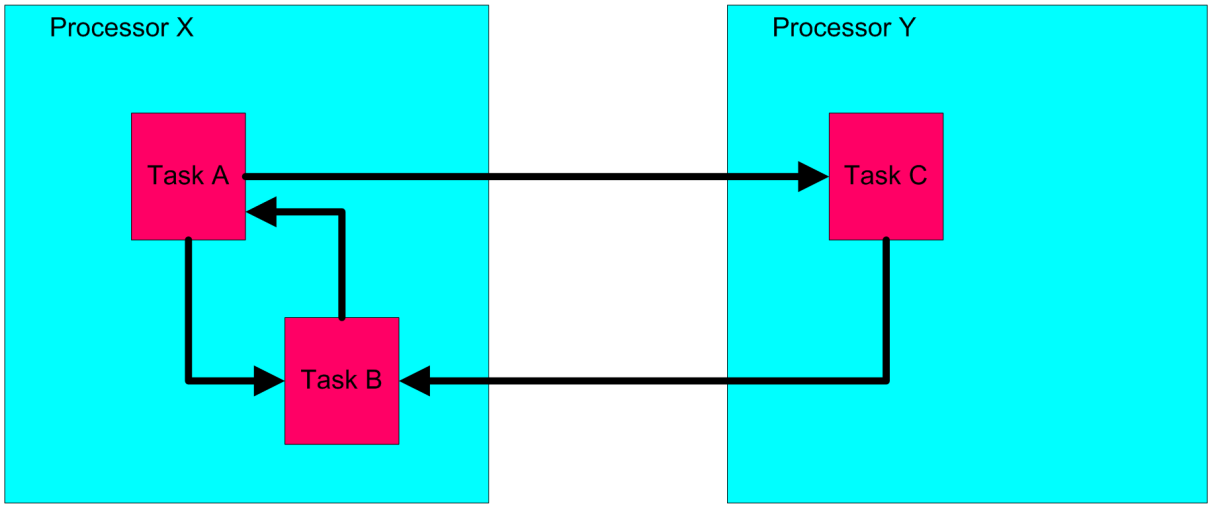
The output from the configurer can be run directly using the server:

- `3L x upper`

More than One User Task

In the previous section we saw how to build a single-task application. In this section, we shall see how we can generalise this method to applications that include more than one task running in parallel.

We continue with the small case-conversion example by splitting the job performed by upper.c into two tasks, as shown in the diagram below. a driver task to handle file I/O, and a processing task which accepts a stream of words containing ASCII character code values on one of its input ports and sends the corresponding upper-case character codes to one of its output ports.



This example is a bit contrived, but splitting a job up into an I/O task and a number of concurrent computation

tasks is commonplace.

The folder `target\C6000\Sundance\Examples\Example-2` contains the C source files for the two tasks, driver.c and upc.c, and a suitable configuration file, upc.cfg. They are repeated here for convenience.

Configuring for More than One Task

The example used here can be found in the folder `examples\example-2`.

Let's start by looking at this configuration file:

```
! Hardware configuration
processor root type=myboard

! Software configuration
task driver ins=1 outs=1
task upc    ins=1 outs=1 data=20k

connect ? driver[0]    upc[0]
connect ? upc[0]    driver[0]

! associate hardware and software
place driver root
place upc    root
```

We have divided the configuration file into three sections, devoted respectively to a description of the hardware comprising the network, the software structure, and the correspondence between the hardware and the software. Here, the hardware is unchanged from the single-task case: it declares the one processor, root.

The software structure is a bit more complex.

Declaring Tasks

This time, we shall need a TASK statement for each of the two tasks in the application. In general, a configuration file must contain a TASK statement for each concurrently executing task in the system:

```
task driver ins=1 outs=1 data=?
task upc    ins=1 outs=1 data=20k
```

As well as the task's name, the TASK statement must specify the number of input and output ports it has. We saw in "The Diamond Model" that a task has a vector of input ports and a vector of output ports that are used to communicate with other tasks. In our example, as we can see from the diagram above, driver and the upc tasks each have one input and one output port, which they use for communicating with each other. The INS and OUTS attributes of the TASK statements define this. Note that the ports are numbered from 0 upwards.

As we have seen, the TASK statement also includes attributes that specify what strategy the configurer should use for allocating memory to the task. Here we are using the DATA attribute, which results in the stack and heap data areas sharing a single area of memory. In the upc task, we specify that this area should be 20k octets in size. The driver task, which has the data=? attribute we have seen before, will get all the free memory remaining once upc's requirements have been fulfilled. Only one task on each processor can have its memory requirements left unspecified in this way. The configurer would otherwise have to decide how to split the remaining memory between several tasks with unspecified requirements. Because an even split is unlikely to be desirable in practice, the configurer refuses to make the decision for you. "Memory Use" gives hints on estimating memory requirements when multiple tasks must be placed on the same processor.

Making Connections between Tasks

The CONNECT statement establishes a channel between two tasks, by connecting an output port to an input port. A channel corresponds to one of the thin arrows in the diagram above, and data can move along it in only

one direction. This means that we need two CONNECT statements to create two channels for bi-directional communication between driver and upc:

```
connect ? driver[0]    upc[0]   ! driver -> upc
connect ? upc[0]    driver[0]   ! upc -> driver
```

The CONNECT keyword can be followed by an identifier naming the connection, but all the configuration statements which declare new identifiers allow a question mark to be used in place of the identifier being declared. This is useful when there is no need to refer to an object after it has been declared. The identifier declared by a CONNECT statement can be used later in a program to simplify access to the associated channels.

After the identifier (or question mark) we code first the output port, and then the input port. Thus, the first CONNECT statement in the example above makes a channel from driver's output port 0 to upc's input port 0.

Consider the following statement:

```
connect ? first[2] second[6]
```

This would allow the following matching statements in the two tasks (assuming the standard declaration of the parameters to main):

```
first:   chan_out_message(12, buffer, out_ports[2]);
second:  chan_in_message(12, mybuffer, in_ports[6]);
```

Assigning Tasks to Processors

Tasks are assigned to processors by the PLACE statement. In our example, both driver and upc are to run on the root processor:

```
place driver root
place upc    root
```

Inter-Task Communication

Now let's consider the code for the two user tasks, and look in particular at the way they communicate.

```
// driver.c  file I/O for upper-casing example

#include <chan.h>
#include <stdio.h>

main(int argc, char *argv[], char *envp[],
     CHAN * in_ports[], int ins,
     CHAN *out_ports[], int outs)
{
   int c;
   for (;;) {
      c = getchar();
      chan_out_word(c, out_ports[0]);
      if (c == EOF) break;
      chan_in_word(&c, in_ports[0]);
      putchar(c);
   }
}
```

Coding the driver task in C is easy. Instead of using the `toupper` function from `ctype.h` as before, it converts characters to upper case by sending a message containing the ASCII character code to the "computation" task and waiting for a reply message containing the result.

C tasks send messages using the channel I/O functions. The `chan.h` package provides functions to send and receive messages of any length. In our program, we use `chan_in_word` and `chan_out_word` to handle word-sized messages. A word is the same size as an int.

The statement in driver.c, which sends character codes to the processing task, is:

```
chan_out_word(c, out_ports[0]);
```

The word (int) value to be sent is passed as the first argument in the function call.

The second argument to `chan_out_word` identifies the output port to which the message is to be sent. out_ports[0] corresponds to output port 0 of the driver task. The CONNECT statement in the configuration file above which refers to driver[0] specifies which task the port is connected to. Here it is the processing task, upc.

out_ports is a vector of pointers to channels, passed into the task via the argument list of its C `main` function. This vector is declared as:

```
CHAN *out_ports[];
```

CHAN is the channel data type defined in the library header file `chan.h`, which must be included by C files that use the channel I/O functions. Each port (i.e., each element in the vector) has type "pointer to channel".

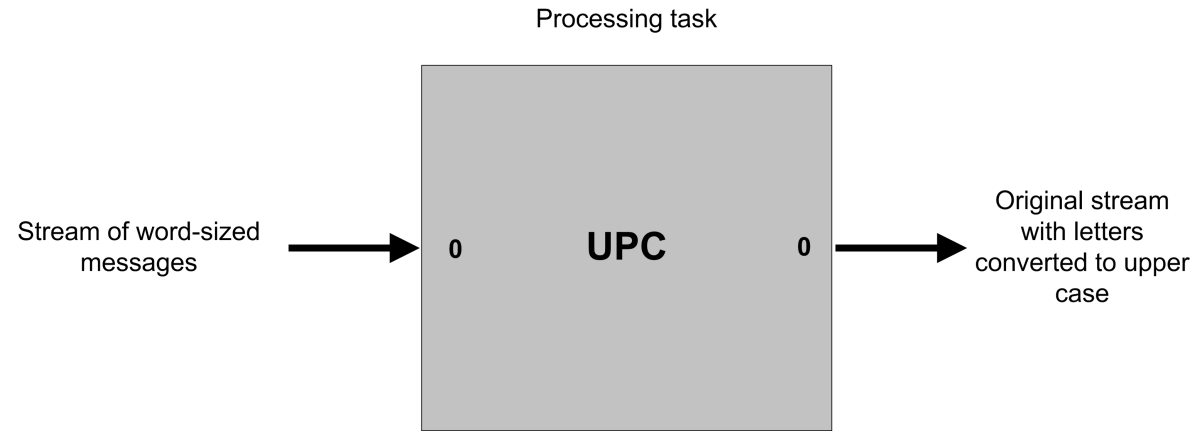
The number of output ports in the vector is defined by the OUTS attribute of the TASK statement used to declare the task in the configuration file. Our driver task has outs=1, so there is only one element in its output port vector, numbered 0.

The value of OUTS is passed into the task as an argument to `main` along with the port vector. It is declared (int outs) in driver.c but not used. It can be used to write tasks that handle an arbitrary number of ports, like the multiplexor task described later on in this chapter.

The `main` function's argument list also provides access to the input port vector in a similar way. In the driver example, the input port vector is given the name in_ports and will have ins elements.

The driver task will keep reading characters from the standard input stream (getchar), sending them to the processing task and writing the reply messages (the translated characters) to the standard output stream until EOF is read.

The next thing to look at is the processing task. It is logically a "black box" with one input port and one output port:



The processing task uses the same channel I/O functions as the driver to send and receive messages. It terminates when it receives an EOF from the driver.

```
// upc.c stand-alone processing task;  communicates with driver.c
   #include <chan.h>
   #include <ctype.h>
   #include <stdio.h>     // for EOF

   main(int argc, char *argv[], char *envp[],
   CHAN * in_ports[], int ins,
   CHAN *out_ports[], int outs)
   {
      int c;
      for (;;) {
         chan_in_word(&c, in_ports[0]);
         if (c == EOF) break;   // terminate task
         chan_out_word(toupper(c), out_ports[0]);
      }
   }
```

Beware when using the channel I/O functions that sending and receiving tasks must always agree on the size of messages. For example, if a task sends a 5-word message, the receiving task must read it as one 5-word unit; it is not possible for the receiving task to read five separate 1-word messages. Trying to do so may cause the processor to lock up or behave unpredictably.

Building a Multi-Task Application

You can build the two-task upper-casing example by compiling, linking using the **`3L t`** command, and then running the configurer as we have already discussed. This uses the full library (rtl.lib) and will make all the facilities of the Diamond library available to both tasks.

Our processing task has no need of this full functionality as upc does no standard C I/O; the header file `stdio.h` is included only for the definition of EOF. A special stand-alone version of the library (sartl.lib) is provided for cases such as these. You can link a program with the stand-alone run-time library by using the **`3L ta`** command in place of **`3L t`**.

Even though all tasks can communicate with the host, linking with the stand-alone library can be worth doing as linking with the full library has costs:

- The resulting task image file contains a large amount of code to handle communicating with the server;
- The task has to initialise its connection with the server and read the command-line string from the host, even if the program does not explicitly use any standard I/O functions like `printf`;
- The configurer will add extra system tasks to any processor on which you place tasks linked against the full library.

The stand-alone library is similar to the full library except that it omits all the functions that require server support (functions for file I/O, in particular). The descriptions of the run-time library functions indicate which functions are members of the stand-alone library.

The complete commands to build and run the two-task upper-casing example are shown below:

```
► 3L c  driver.c
► 3L c  upc.c
► 3L t  driver
► 3L ta upc
► 3L a  upc.cfg upc.app
► 3L x upc
```

Stop the application by typing Ctrl-Z, which the driver task will read as an end-of-file.

You can try this out for yourself, using copies of the relevant files, which are supplied in the Diamond kit. You will find them in the folder `target\c6000\Sundance\examples\example-3\`. A variant of this code that accesses the input and output channels by name rather than by port indexes may be found in `target\c6000\Sundance\examples\example-3b\`. There is also a batch file, upc.bat, for building the application.

Shutting Down Cleanly

In simple applications that have only one task linked with the full library, the server shuts down when this task exits. You can then run another application.

When an application contains more than one task linked with the full library, the server will not stop until they have all exited. However, it does not wait for tasks that are linked with the stand-alone library to exit.

The simple examples discussed up to now have not had to deal with errors. When an individual task detects an error, the following choices are available:

- The offending task may simply exit. The rest of the application will continue to operate, except that attempts to communicate with the failed task will hang. This is the simplest approach, and is recommended for new users. The drawback is that the host server will continue to run even when the application it is serving has failed, and may have to be shut down manually.
- A failing task may orchestrate a controlled shutdown of all tasks linked with the full library, by sending messages to them. This approach is harder to design and implement, but results in the host server terminating cleanly.
- A task may call the library function `_server_terminate_now`, which forces the host server to stop immediately. This approach should be avoided if possible, because other tasks will not be notified of the shutdown and may have important open files, which will not be closed properly.

Scheduling

The Diamond model for scheduling threads and tasks was outlined earlier. Here we shall go into a little more detail.

Scheduling is done by the microkernel. Each thread has a priority. There are eight levels of priority, numbered 0–7, level 0 being the highest priority and level 7 the lowest. Level 0 is referred to as urgent. The header `thread.h` defines the literals `THREAD_URGENT` and `THREAD_NOTURG` that correspond to priority levels 0 and 1 respectively.

Once a thread is running, it will only be suspended for one of the following reasons:

1. It calls the `thread_deschedule` function, which voluntarily interrupts execution.

2. It has to wait for some event external to the thread. This could be, for example:
   - waiting for input or output to end, whether via a channel or a link;
   - waiting for a semaphore to be signalled;
   - waiting for an event to be signalled or pulsed;
   - waiting for a timer event;
   - waiting for an `alt_wait` function call to end.

3. It is pre-empted, that is, it is paused by the microkernel so that a higher priority thread, which is now ready to execute, can be started instead.

4. It has used up its time-slice, that is, it has been executing without interruption for a certain length of time.

Urgent threads will only be suspended for the first two of these reasons; they will never be pre-empted or time-sliced.

Once a thread has been suspended, the highest-priority thread that is waiting to run will be started. If a thread uses up its time-slice, another thread of the same priority will be given a turn, if one is available, so that the time is shared evenly between threads of the same priority; if none is available, the time-sliced thread will continue to run.

If there are no threads ready to be run at all, the idle thread will run. This executes the IDLE instruction and waits for an interrupt.

When you create a new thread using one of the functions in this section, you have to specify the priority at which the thread is to run. The temptation to create every thread with `URGENT` priority should be resisted, as urgent threads cannot be pre-empted or time-sliced. Once started, an urgent thread will run until it voluntarily deschedules itself or has to wait for an external service. This means that all other threads, including all other urgent threads, will be prevented from executing at all. These "locked out" threads may include parts of the communication software, and this could have a bad effect on the performance of the application as a whole.

Multi-Processor Applications

If you have followed the examples this far, the generalisation from a multi-task system running on a single processor to a full multi-processor system will be fairly obvious. All that is required is a change to the configuration file to describe the extra hardware and place some tasks onto processors other than the root processor.

Configuring Multi-Processor Applications

You can find the source code for the example discussed here in the folder `target\c6000\Sundance\examples\example-3\`.

The new configuration file is shown here:

```
! Hardware configuration
processor root  DEFAULT
processor addon DEFAULT
wire ? root[0] addon[4]

! Software configuration
task driver ins=1 outs=1
task upc    ins=1 outs=1 data=20k

! Channel connections
connect ? driver[0] upc[0]
connect ? upc[0] driver[0]

! Task placement
place driver root
```

```
place upc    addon
```

We are going to run the case-conversion example on a two-processor system with the driver task on the root processor and the upc task on the other processor. The extra hardware must be declared in the configuration file:

```
processor addon DEFAULT
wire ? root[0] addon[4]
```

The first line here gives a name (addon) to the second processor. We must also specify its type, depending on what kind of board it is. Here we have assumed that a default processor type has been defined.

The second line contains a new statement, WIRE. This declares that there is a physical connection between link 0 of the root processor and link 4 of the new processor, addon. Obviously, this WIRE statement must reflect the real hardware configuration. There must be at least one path of WIREs from the root node to every processor in a network. This path may pass through any number of intermediate nodes.

If we reconfigured the application with only these changes to the configuration file, the addon processor would be unused because the upc and driver tasks are both placed on the root processor. We can fix this by modifying the PLACE statement for upc

```
place upc addon
```

Now the configurer will arrange for the code of the upc task to be loaded into the second processor when the complete application is started with the server.

Notice that the CONNECT statements do not need to be changed, as the logical connections between the tasks are the same as before. If CONNECT is used to connect ports of two tasks on the same processor, the channel the configurer binds to the two ports will be implemented by a memory copy. If the tasks are on different processors, the channel will involve the interprocessor links and Diamond's communication software. As far as the programmer is concerned, these two cases are identical, and exactly the same library functions are used(`chan_out_word`, `chan_in_word` and so on).

It's important to understand the difference between the WIRE statement and the CONNECT statement. WIRE specifies the actual hardware connections between processors; CONNECT specifies the logical connections (channels) between tasks. The index values in WIRE statements refer to link numbers; those in CONNECT statements refer to elements of the input and output port vectors. If a CONNECT statement requires messages to be transferred to another processor, the communication software makes use of the available WIREs to do this. The programmer does not need to know which WIRE, or which hardware link, is used.

Links and Channels

It may be worthwhile at this moment to re-emphasise the difference between a channel and a link.

A **`link`** is a physical, bi-directional connection between two processors, which allows them to communicate. This can be a TI-style comport, some other hardware-supported link, or a section of shared memory.

A **`channel`** is a uni-directional means for tasks to communicate, but these tasks need not be on different processors. Communication with a task on another processor through a channel will imply the use of links; they are employed to support the channel connection. Communication with a task on the same processor through an internal channel will not make use of links.

As we have seen, a WIRE statement describes a link which joins two processors, whereas a CONNECT statement describes a channel connecting two tasks.

Virtual Channels

By default, if the tasks joined by a CONNECT statement are on different processors, the configurer will attempt

to use a virtual channel to connect the two tasks. Virtual channels are very flexible:

- Messages sent to tasks on distant processors are automatically forwarded via intermediate network nodes;
- A single WIRE can support any number of virtual channels;
- Virtual connections between tasks are not limited to the available physical links between processors.

In short, any task can communicate with any other task, on any processor, irrespective of the physical network layout. This flexibility must clearly come at a price; a certain amount of software overhead is required to implement virtual channels.

The throughput of a virtual channel may be up to 50% of a physical channel for long messages sent to a neighbouring network node. Performance drops for shorter messages, down to about 25% of a physical channel for messages of 1000 words and below 10% for short messages of fewer than 100 words. Performance also drops when messages must be forwarded through intermediate nodes. Note that these overheads are to a large extent inherent in any software message-routing system. It is possible to tune virtual channel performance for your application.

There is a current limitation in the use of virtual channels: a virtual channel can only connect two processors of the same general type, for example C6000 processors or C4x processors. In addition, there must be a path of wires between these processors that only passes through other processors of the same type. For example, you cannot have a virtual channel between ROOT and N3 in the following example:

```
PROCESSOR ROOT TYPE=C6000
PROCESSOR N1   TYPE=C40
PROCESSOR N2   TYPE=C6000
PROCESSOR N3   TYPE=C6000

WIRE ? ROOT[0] N1[3]
WIRE ? N1[0]   N2[3]
WIRE ? N2[0]   N3[3]
```

However, if you added another WIRE from ROOT to N2, you would be able to have a virtual channel between ROOT and N3.

Physical Channels

Diamond also allows CONNECT statements to be mapped directly onto physical channels (WIREs) when necessary, completely eliminating the extra overhead. Communication performance is unlikely to be a big problem for our upper-casing example, but if we did want to use physical channels for the connections between the driver and upc tasks, we could change the two CONNECT statements as follows:

```
connect ? driver[0]    upc[0] physical
connect ? upc[0]    driver[0] physical
```

There is no need to re-compile or re-link either of the tasks.

You may also specify explicitly that particular channels must be virtual:

```
connect ? driver[0]    upc[0] virtual
connect ? upc[0]    driver[0] virtual
```

If a CONNECT statement does not specify either VIRTUAL or PHYSICAL, the configurer will make it virtual by default. Putting one of the following statements anywhere in a configuration file can change this default setting:

```
default connect physical
default connect virtual
```

The configurer also provides a command-line switch to make all connections physical by default:

```
3L a -p …
```

The configurer will report any conflicts among these different ways of specifying default behaviour.

Physical Channel Restrictions

When using physical channels, take care that enough WIREs are available for all the inter-processor connections required. Each WIRE can support only two physical channels, one in each direction. If two processors need to communicate over a physical channel, they must be directly connected by at least one physical WIRE.

"Restrictions on Network Configuration" and "Restrictions on Physical Channels" describe in detail the restrictions on the use of physical channels. Some restrictions on the use of virtual channels are also described there. These can usually be ignored in networks that use only virtual channels, but become more important when physical channels compete for the available WIREs.

WIRE Usage by Virtual Channels

A virtual channel is implemented by a software mechanism that needs communication in both directions along a wire. It follows that a virtual channel cannot use a wire that is supporting any physical connections at all.

Error Detection on Virtual Channels

The size of each message sent using chan_out_word and the other message-passing calls must exactly match the size expected by the receiver. Failure to follow this rule for physical channels, or for internal channels between tasks on the same processor will usually result in the application hanging up or crashing. One benefit of virtual channels not mentioned previously is that the Virtual Channel Router (VCR) is able to check for this type of error. When a mismatch occurs on a virtual channel, the server will report the error as a software exception like this:

```
*** Software exception: 00001202 00000008 00000044
Processor=0 Severity=error
Group=3L Facility=VCR (Virtual Channel Router)
```

Here, processor 0 wanted to send 8 octets but the receiver wanted to take 44_{16}.

```
*** Software exception: 00001102 00000044 00000008
Processor=1 Severity=error
Group=3L Facility=VCR (Virtual Channel Router)
```

Here, processor 1 wanted to receive 44_{16} octets but the receiver wanted to send 8.

Note that two errors may be reported, one from each end of the channel. Error 1102_{16} is from the sending end; 1202_{16} comes from the receiver. Attempting to send or receive a zero-length message is also reported as an error.

To find out which named processor corresponds to the processor number in a software exception message, you need to use the configurer's "map" option. For example:

```
► 3L a -m upc.cfg upc.app
```

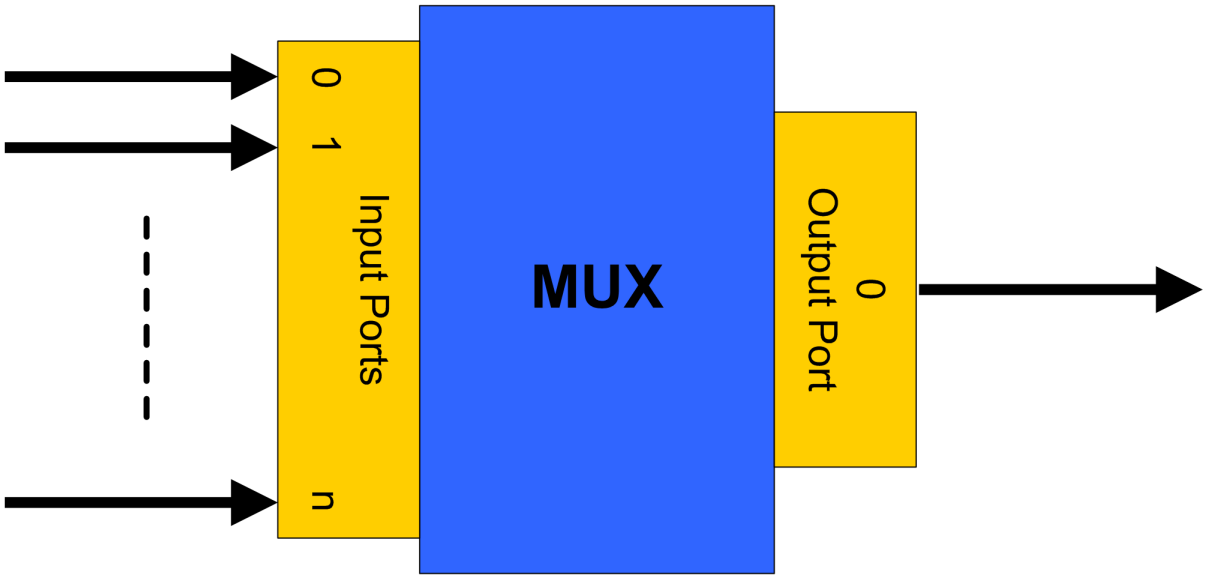
The map is written to the standard output stream and looks like this (the processor names are those assigned in your configuration file):

```
UPR node 0 is processor "root"
UPR node 1 is processor "addon"
```

Armed with the configurer's map output, we can now interpret the processor numbers from the software exception messages we saw above. This particular error arose because a task on the "addon" processor tried to send a 17-word (68 octet) message to a task on the "root" processor, which was expecting to receive only two words (8 octets).

Simultaneous Input

One thing we have not yet seen how to do is to wait for a message from any one of a number of concurrently executing tasks. For example, a multiplexor task that accepted messages on any of an arbitrary number of input ports and passed them on through a single output port would be a useful building block.



A task connected to the output port of the mux task sees a sequential stream of messages, even though they are coming from any number of input tasks, in any order.

The problem in implementing mux is that we cannot simply issue channel read requests on the input channels in order. This is because a channel read will not terminate until the data have actually arrived and been read. If the channel we pick has no message waiting to be read, nothing will happen. In the meantime, all the other channels will be held up. What we need is a way to identify which channel has an incoming message waiting to be read. We can then issue a read on that channel alone.

The `alt.h` functions provide this facility.

Here, we give the flavour of these functions by showing a Diamond implementation of the multiplexor task that uses the `alt_wait_vec` function to wait for a message to arrive from any element of an array of (pointers to) channels. The multiplexor task's input port vector is just such an array of pointers to channels, so it can be passed directly to `alt_wait_vec` along with a count of the number of elements in the array.

`alt_wait_vec` waits for a message to become available on any of the channels pointed to by the array, in this case any of the multiplexor task's input ports. It then returns the index in the array of the channel that is ready to read a message. If more than one channel becomes ready at the same time, the system will choose which one to handle first. If no channel ever becomes ready, the function will never return.

Once `alt_wait_vec` has determined the channel on which a message is incoming, the rest of the mux task is quite straightforward. First, read the message from that channel into a buffer, then echo the message to the single output port. In the example, the messages consist of a fixed length (one word) header giving the size of a trailing variable-length part. Only one message buffer is required no matter how many input ports are connected to the multiplexor task. Messages arriving on any other channels are blocked while the multiplexor deals with the current message.

```
// Examples\General\altmux.c
// message multiplexor using 'alt' package

#include <alt.h>
#include <chan.h>

main(int argc, char *argv[], char *envp[],
```

```
      CHAN *in_ports[], int ins,
                                CHAN *out_ports[], int outs)
{
   int msglen;      // number of words in message
   char buf[1024];  // message buffer

   for (;;) {       // read messages forever
      //
      // wait till next message received on any input port
      //
      int i = alt_wait_vec(ins, in_ports);

      //
      // read the message from that port
      //
      chan_in_word(&msglen, in_ports[i]);
      chan_in_message(msglen, &buf[0], in_ports[i]);

      //
      // ...and copy it to the single output port
      //
      chan_out_word(msglen, out_ports[0]);
      chan_out_message(msglen, &buf[0], out_ports[0]);
   }
}
```

Multi-Threaded Tasks

Creating Threads

The `<alt.h>` functions allow a limited amount of parallelism to be introduced into a sequential task. Diamond also allows tasks to be fully multi-threaded. This means that a task can contain any number of concurrent processes each of which is independently executing the code of the task. All the threads in a task share the same static, extern and heap data. The threads can still operate independently because each one is given its own stack for auto variables.

When it starts, each task has just one thread, its main function. New threads are created dynamically by calling the library function thread_new. All of the library functions discussed in this section are described more fully in the library description and the list of functions.

A thread terminates when its initial function returns.

Waiting for Threads to Finish

The thread creation functions all return a handle value. A different thread can use this handle with the function `thread_wait` to wait for the thread to terminate:

```
THREAD_HANDLE h = thread_new(MyThread,…….);
...
thread_wait(h);  // wait until the thread stops
// (MyThread returns).
```

Access to Shared Data

If multiple threads in a task are operating on shared data, say a buffer held in static storage or on the heap, they must synchronise their access to the data. Threads can synchronise their operations using either semaphores or channels.

Note that variables shared between threads may need to be declared as **`volatile`** to make sure that changes made by one thread are visible to other threads. Consider what would happen if the compiled code of a thread function were to keep a shared variable in a register. Changes made by other threads to the value of that variable in memory would be invisible. Using volatile prevents the compiler from keeping a variable in a register.

Synchronisation Using Semaphores

Semaphores may be used to synchronise the operation of user threads. To illustrate the use of semaphores we have recoded the multiplexor example presented previously to use multiple threads interlocked by a semaphore in place of the `<alt.h>` functions.

A new execution thread is created for each input port. Each thread does a simple sequential read and waits for a message. As soon as one thread receives a message it waits until a semaphore indicates the output port is free. Using a semaphore prevents disaster if two threads each try to write to a shared object like the output port at the same time.

The following code fragment shows the semaphore version of the multiplexor task. This implementation shares one message buffer area between all its threads as well as sharing the output port. All of a task's threads share the same static, extern and heap data. Each thread has its own stack for auto variables, so each thread in the example has its own msglen variable. The stack space for a thread is created automatically (from the heap) by the `thread_new` function. Any number of input threads may have read the length part of their incoming messages, but the buf_free semaphore ensures that only one is using buf and out_ports[0] at any time.

```
// Examples\General\Mmux.c
// message multiplexor task

#include <chan.h>                           // required header files
#include <thread.h>
#include <sema.h>

char buf[1024];
SEMA buf_free;                              // controls access to buf
CHAN **in_p, **out_p;                       // global pointers to
                                            // port vectors
void receive(void *p)                       // handle one input port
{
   int msglen;                              // one for each thread
   int i = (int)p;                          // i = port to service

   for (;;) {                               // forever...
      chan_in_word(&msglen, in_p[i]);       // await message
      sema_wait(&buf_free);                 // wait until buf free
      chan_in_message(msglen,               // read message into
                      &buf[0],              // the shared buffer
                      in_p[i]);             // from our port
      chan_out_word(msglen, out_p[0]);      // send to out_ports[0]
      chan_out_message(msglen, &buf[0], out_p[0]);
      sema_signal(&buf_free);               // let someone else in
   }
}

main(int argc, char *argv[], char *envp[],
     CHAN *in_ports[], int ins, CHAN *out_ports[], int outs)
{
   int i;

   sema_init(&buf_free, 1);                 // initially free

   in_p = in_ports;                         // make ports...
   out_p = out_ports;                       // globally available

   for (i=0; i < ins; i++) {                // one thread per input
      thread_new(receive,                   // function
                 1024*sizeof(int),          // stack size in bytes
                 (void *)i);                // port to use
   }
```

```
}
```

Any function that is going to be started by `thread_new` must have exactly one argument, of type void *. Notice that to pass the number of the input port to each receive thread, we have had to cast it to a void *, and then cast it back to an int inside the thread.

Warning

At first glance, the alternative seems to be to pass a pointer to the variable i:

```
for (i=0; i < ins; i++) {
   thread_new(receive, 1024*sizeof(int), &i);
}
```

This is not appropriate, as we cannot predict when the new threads will start to execute. It is likely that the value of i will have changed by the time a receive thread comes to pick it up.

Note that the main function of our example task returns when it has finished starting the "receive" threads; those threads will continue to execute even though their parent has stopped.

If you haven't used semaphores or a similar method for controlling concurrent access to shared objects before, you should read a good introduction to the subject, such as [Lister] or [Tanenbaum]. It is possible to introduce difficult-to-trace errors into a program if threads forget to synchronise access to a shared object by waiting for a semaphore.

Synchronisation Using Channels

A channel can be used to synchronise two threads by including the header `<chan.h>` and then declaring a static, extern or heap variable of type `CHAN`. Once this channel has been initialised using the `chan_init` function, a pointer to it can be used to specify the channel to be read or written by any of the channel I/O functions.

Remember that each channel can only be used to transmit data in one direction between exactly two threads. You cannot use a channel to transmit data in both directions (you must use two channels) and you cannot allow more than one thread to be waiting for input from the same channel.

There follows a task that creates just two threads, a produce thread that generates a sequence of word-sized messages and a consumer thread that processes them. The messages are transmitted across an internal channel, chan. The channel transmits the data, and also ensures synchronisation: the consumer cannot proceed once it has called chan_in_word until the producer sends a message over chan. Similarly, if the consumer thread is busy when the producer attempts to send a message, it will be blocked until the consumer comes to read its next message.

Caution

Any internal channels used to synchronise the operations of multiple threads must be declared and initialised before those threads are started. Failing to initialise an internal channel is a common mistake.

```
// Examples\General\ProCon.c

#include <stddef.h>
#include <thread.h>
#include <chan.h>
#include <par.h>
```

```
#define STACKSIZE 1024

CHAN chan, consumer_finished;

void producer(void *p)   // generate 10 values
{
   int i;
   for (i=0; i < 10; i++) chan_out_word(i, &chan);
}

void consumer(void *p)   // processes 10 values
{
   int i, val;
   for (i=0; i < 10; i++) {
      chan_in_word(&val, &chan);
      par_printf("%d\n", 2*val);
   }
   chan_out_word(1, &consumer_finished);
}

main()
{
   int dummy;
   chan_init(&consumer_finished);
   chan_init(&chan);
   //
   // channels initialised BEFORE starting the threads!
   //
   thread_new(producer, STACKSIZE, NULL);
   thread_new(consumer, STACKSIZE, NULL);
   //
   // wait for all threads to terminate
   //
   chan_in_word(&dummy, &consumer_finished);
   //
   // before exiting
   //
}
```

The following is a variant of the previous example that uses thread_wait rather than a channel to achieve a controlled termination:

```
#include <stddef.h>
#include <thread.h>
#include <chan.h>
#include <par.h>
#define STACKSIZE 1024

CHAN chan;

void producer(void *p)   // generate 10 values
{
   int i;
   for (i=0; i < 10; i++) chan_out_word(i, &chan);
}

void consumer(void *p)   // processes 10 values
{
   int i, val;
   for (i=0; i < 10; i++) {
   chan_in_word(&val, &chan);
   par_printf("%d\n", 2*val);
}

main()
{
```

```
    int dummy;
    THREAD_HANDLE consumer_thread;
    chan_init(&chan);
    //
    // channels initialised BEFORE starting the threads!
    //
    thread_new(producer, STACKSIZE, NULL);
    consumer_thread = thread_new(consumer, STACKSIZE, NULL);
    //
    // wait for consumer to terminate
    //
    thread_wait(consumer_thread);
    //
    // before exiting
    //
}
```

Threads and Standard I/O

It is a bad idea for the main function of a task to return while any threads it has created are still active if, as in the program above, one of those threads may be using C standard I/O. If this happens, the main function will exit, causing the run-time library to attempt to shut down the I/O system and close all open files before some thread which needs to do I/O has finished. This can lead to several obscure errors, most commonly reports from the server of "protocol error". To forestall this possibility, an extra channel has been added in this example from the consumer thread back to the original main thread. It is used purely for synchronisation. When the consumer thread is about to terminate, it sends a dummy message over this channel. The main thread waits for this message before returning. A semaphore could have been used here instead.

Note that the multiplexor example above demonstrates a occasion where it is safe to let main terminate while other threads are active; those threads do not require access to the host for I/O.

When an application has multiple threads that need to continue running when the main function has finished, you should stop the main thread by calling `thread_stop`.

Synchronising Server References

Note the use of `par_printf` in place of `printf` in the consumer thread above. If multiple threads are active in a task, and more than one thread needs to communicate with the server (usually for I/O), then a semaphore (`par_sema`) must be used as an interlock to ensure that only one thread at a time can interact with the host. The `par.h` header provides ready-interlocked versions of some common functions, such as `printf`. The interlock is not actually necessary in our example, since no other thread will be attempting to use the full library at the same time, but it is as well to be aware of the problem. Note that `par_sema` is also used to synchronise accesses to the memory access functions, malloc and free. Failing to synchronise these accesses can result in obscure errors like "protocol error".

The following shows an example of explicit synchronisation. Two threads use fprintf to write to the same file. You must protect these potentially concurrent calls to `fprintf` by waiting for the `par_sema` semaphore declared by `par.h`. This is just what the built-in function `par_fprintf` does automatically. You must use this technique to protect concurrent calls to server-access functions for which no par_ version is available.

```
// Examples\General\ParEx.c
#include <stdio.h>
#include <thread.h>
#include <par.h>
#include <sema.h>

#define STACKSIZE 1024
FILE *f;
SEMA all_done;

void output(void *base)
```

```
{
   int i;
   for (i = (int)base; i < 10; i += 2) {
      sema_wait(&par_sema);
      fprintf(f, "%d\n", i);
      sema_signal(&par_sema);
   }
   sema_signal(&all_done);
}

main()
{
   f = fopen("out.dat", "w");
   sema_init(&all_done, 0);
   thread_new(output, STACKSIZE, (void *)0);
   thread_new(output, STACKSIZE, (void *)1);
   // wait for both threads to stop
   sema_wait_n(&all_done, 2);
   // before leaving main
}
```

Not all library functions need to be protected in this way. Some functions are thread safe and may be called from any thread without special precautions. The `alt.h`, `chan.h`, `ctype.h`, `link.h`, `par.h`, `sema.h`, `setjmp.h`, `stdarg.h` and `timer.h` functions are all thread safe. Descriptions of functions that are not thread safe are marked Server or Heap.

Note that it is only multiple threads within the same task that must explicitly synchronise their calls to the server. Synchronisation of threads within separate tasks is automatic.

Threads versus Tasks

Threads are "lightweight" processes:

- They share their code, heap, static and external data memory with all the other threads created by the same task;
- They can share data and may communicate either via shared memory or by using channels;
- All the threads of a single task run on the same processor, allowing them to share memory.

Tasks on the other hand are more substantial:

- They only communicate via channels;
- Each task has its own code and data areas, separate from all other tasks; code, including run-time library functions, is not shared between tasks, even tasks placed on the same processor;
- A task can be moved to a different processor simply by reconfiguration.

Two operations to be performed concurrently can be usefully performed by threads rather than tasks if all of the following conditions hold:

- They will never need to be run on distinct processors;
- The operations are closely coupled, i.e., they share a lot of common code. Code is automatically shared between threads, but each task has its own copy of all of its code, including library functions, so that if necessary it can later be moved to a different processor without requiring recompilation or re-linking;
- The operations logically operate on shared data structures. This may be more efficiently performed directly by concurrent threads than by tasks copying the data back and forth as messages when they are modified.

Using Assembly Language

The effort of hand-coding an inner loop in assembly language can sometimes be worthwhile when performance is critical. The Optimizing C Compiler[C Compiler] contains information about register usage and function

argument-passing conventions, which you will need if you want to write assembly-language functions callable from Diamond.

Assembly language is also used for writing low-level interrupt handlers.

Part II. General Reference

Standard Diamond Features

Table of Contents

Chapter 6. Commands

Diamond applications are built and executed using a variety of compilers, linkers, and other utilities, many of which will change from edition to edition. Combinations of different hardware targets and board manufacturers will give rise to numerous command variations.

In order to simplify this and allow project to be built for multiple targets, Diamond includes a command utility, `3L.exe`, that hides much of the complexity. You can extend this utility to adapt it to your working environment.

Command Syntax

The general form of a command is:

```
3L options function arguments
```

options	zero or more items from the following list:	
	-H	Display help information
	-T target	Define the target processor, as described below.
	-V	Verbose operation. Display information about the execution of the command. This is useful for debugging new functions.
function	the function to be executed, as described below	
arguments	the arguments to be passed to the selected function	

Functions

The command utility supports an extensible set of command functions; uppercase and lowercase letters are equivalent. The standard set is as follows:

C	Compile a source file
T	Create a task by linking a number of object files against the full library.
TA	Create a stand-alone task by linking a number of object files against the stand-alone run-time library.
A	Build (configure) an application under control of a configuration file. You should not give a target for this command.
X	Execute an application using the server. You should not give a target for this command.

This list may be extended in particular editions to support different processor types. For the Sundance C6000 edition, the following extra commands are available:

C67	Compile a source file for a floating-point C6000.
T67	Create a task by linking a number of object files against the full library and the TI C67 floating-point library.
T67A	Create a stand-alone task by linking a number of object files against the stand-alone library and the TI C67 floating-point library.
C64	Compile a source file for a member of the C64 family of processors.
T64	Create a task by linking a number of object files against the full library and the TI C64 library.
T64A	Create a stand-alone task by linking a number of object files against the stand-alone library and the TI C64 library.

Targets

A target is a way of identifying the particular combination of processor type and board manufacturer that is relevant to a compile or link function. Command definitions are held in one or more files named `Command.dat`, and a target helps the command utility to locate the correct one.

There are three types of target:

Processor	The type of processor for which a task is to be built. Examples of this could be c6000 or ppc.
Manufacturer	The manufacturer of the target hardware, for example, Sundance, or Anon.
Processor/Manufacturer	Where the processor or the manufacturer alone does not uniquely define the target hardware, you can give both. For example, c6000/Sundance, ppc/Sundance, or ppc/Anon.

The command utility searches the Diamond installation folder structure for **`command.dat`** starting at a subfolder determined by the target. If you provide a target, the subfolder will be `bin\`*`target`*, otherwise it will be `bin\`. Finding more than one command file within the chosen folder is considered an error and the utility will fail with an message. This means that you must provide a target when you have more than one edition of Diamond installed. If you only have one, you can always omit the target specification.

Typical commands are as follows:

► 3L c driver.c	Compile driver.c.
► 3L c upc.c	Compile upc.c.
► 3L t driver	Create driver.tsk by linking against the full library.
► 3L ta upc	Create upc.tsk by linking against the stand-alone run-time library.
► 3L a upc	Create upc.app by running the configurer.
► 3L x upc	Execute upc.app with the Windows Server.
► 3L –T xxx c y.c	Execute the c command for the processor or manufacturer xxx.

Adding Your Own Functions

This section describes how you can extend the command utility by adding your own functions. This can be useful if you have definitions or switches that you invariably add to your commands. Rather than modify existing commands, we recommend that you add new functions to the end of the standard list.

Command File

Commands are defined in a file called `command.dat`. There is a version of this file for every edition of Diamond you have installed. You can find the file in the folder:

```
bin\<processor>\<manufacturer>
```

As an example, if you have installed the Sundance c6000 edition of Diamond in the standard place, this would be: `c:\3L\Diamond\bin\c6000\Sundance\`

The file comprises a number of lines of text. If a line becomes inconveniently long, you can break it at any point by inserting a backslash and a newline. Any line starting with `//` is treated as a comment and is completely ignored.

Function Name

A function is defined by a sequence of lines, where the first line gives the name of the function and any aliases you wish to give it. Each name must start with a minus sign. For example:

```
-CDEBUG, -CD
```

This would introduce the definition of a function called CDEBUG. The function could also be invoked by using the name CD.

Operations

The subsequent lines in the file define the operations needed to perform the function. The list of operations is terminated by another function definition or the end of the file.

There are three types of operation:

Execute fn arguments	The function fn is executed as a command with the given arguments.
File filename	A new text file, `filename`, is created. All the subsequent lines in the `Command.dat` file that start with a colon are written to the file (leading whitespace and the colon are not written). Writing to the file stops as soon as a line that does not start with a colon is reached.
Delete filename	filename is deleted.

Macros

You can use several macros within the lines making up a function's operations. They are replaced by strings computed by the command utility.

@@	is replaced by the single character @.
@1	is replaced by the first item in the 3L command's arguments.
@A	is replaced by the whole of the 3L command's arguments.
@D	is replaced by the definition of the environment variable Diamond. It gives the path of the Diamond installation folder and ends with a backslash.

Example

This is an example of a function that is the same as the standard C function (compile) but also defines the symbol "DEBUG".

```
-CDEBUG, -CD
Execute cl6x -qq -I"@Dbin\c6000\Sundance\include" \
--symdebug:coff -O3 -c -pdsw225                   \
-dDEBUG                                           \
@A
```

You can test this function by running 3L.exe in verbose mode:

```
► 3L -V CD hello.c
Diamond:     C:\3L\Diamond\
Command:     CD
Parameters:  hello.c
Command file C:\3L\Diamond\bin\c6000\Sundance\Command.dat
Execute cl6x -qq -I"C:\3L\Diamond\bin\c6000\Sundance\include"
--symdebug:coff -O3 -c -pdsw225  -dDEBUG hello.c

►
```

Chapter 7. Configuration Language

The 3L configuration language is the language used to write configuration files for the various 3L configuration utilities. It is designed to allow easy description both of physical processor networks and of user applications built up out of tasks, without the user being concerned with the details of how the tasks are actually loaded into the processor network.

Each of the configuration utilities will, in general, accept a subset of the language described here, according to its needs. In addition, implementations for different processors will differ semantically to some extent. You should refer to the discussion of the configurer on the C6000.

Standard Syntactic Metalanguage

In the rest of this chapter, we shall be describing the syntax of the configuration language. To make sure that this description is precise and unambiguous, we shall make use of a metalanguage, that is, a language specifically designed for describing other languages.

The metalanguage we shall use is the one specified by British Standard 6154. Many readers will be familiar with similar metalanguages, particularly those of the well-known Backus-Naur family, so here we shall give only a brief description of BS6154. For more detail, you should consult the standard itself. There is also a tutorial introduction, which is available from the National Physical Laboratory.

- Terminal strings of the language—those not built up by rules of the language—are enclosed in quotation marks.
- Non-terminal phrases are identified by names, which may consist of several words.
- A sequence of items may be built up by connecting the components with commas.
- Vertical bars ("|") separate alternatives.
- Optional sequences are enclosed in square brackets ("[" and "]").
- Sequences that may be repeated zero or more times are enclosed in braces ("{" and "}").
- Each phrase definition is built up using an equals sign to separate the two sides, and a semi-colon to terminate the right hand side.

Configuration Language Syntax

The lower-level syntax of the configuration language deals with the way in which multiple input files are handled, with comments and with line continuation. This topic is treated informally below.

"Constants and Identifiers" and "Statements" cover the higher-level syntax, which deals with how the tokens and statements of the language are built up. To help with this, we shall use the standard syntactic metalanguage, but with an additional simplification to make the syntax more readable. To show this, consider the following syntax rule written in the BS6154 metalanguage:

```
example rule = "first", "second";
```

Interpreted strictly, this rule would be satisfied only by an input text that read "**`firstsecond`**" . In the syntax presented here, it should be taken to match "**`first`**" followed by "**`second`**" , but in such a way that the two items are distinguishable. For example, a space character in the input file might separate the two words here. When the two items are distinguishable in the input file without a space between them, then they may be abutted, as in the following example:

```
second example rule =  "first", "=";
```

Valid input text for this rule could be, for example, "**first=**" or "**first =**" .

Low-Level Syntax

The general form of a configuration specification is designed to be as simple as possible to use. The following example shows the ways in which the formatting, commenting and continuation facilities available in the configuration language can be used:

```
! this is an example of a comment
! a blank line follows...

! next, a statement continuation...
PROCESSOR -
  host

! now, both features in combination...
PROCESSOR - ! comment AND continuation
root
```

The above sequence is, to the configurer, exactly equivalent to the following:

```
PROCESSOR HOST
PROCESSOR ROOT
```

The various facilities used above can be summarised as follows:

- The case of letters is not significant, except in a string constant. In all other contexts, upper and lower case letters may be used interchangeably.
- White space within a line (space characters, tab characters and so forth) is compressed; for example, three consecutive spaces would be seen as one.
- Everything from an exclamation mark character "!" to the end of the line is taken to be a comment, and is discarded.
- If the last non-whitespace character on a line is a hyphen "-", the line is taken to be continued onto the next line.
- Continuation and commenting can be used together; the hyphen must then be the last non-whitespace character before the comment.

Certain statement types (TASK, PROCEDURE, and PROCTYPE) often have several attributes. The complete list of attributes may be enclosed in braces, in which case newlines are ignored and no continuations are needed. For example, the following two TASK statements are equivalent:

```
TASK Fred  ins=1  -
           outs=2 -
           data=16K

TASK Fred {
   ins=1
   outs=2
   data=16K
}
```

In addition to these line formatting considerations, note that the configurer can accept any number of input files rather than simply one. This facility is designed to allow different parts of the description of an application to be held in separate files. For example, the description of the physical network might be held in one file and the description of the user's application in another. The configurer simply treats each input file in order as part of one long input stream.

Constants and Identifiers

Numeric Constants

Several different kinds of numeric constant are available to meet the different uses of constants within the configuration language. For example, a constant may be expressed in decimal notation or in hexadecimal.

A special notation is provided to extend the decimal constant with a scaling letter; this is most commonly used in specifications of memory allocation. The scaling letters "K" and "M" scale the decimal constant they follow by 1024 and 1024×1024 (1048576) respectively. Note that it is not possible to add a scaling letter to a hexadecimal constant; the configurer would interpret such a combination as the hexadecimal constant followed by a single-character word containing the scaling letter.

Although all numeric constants in the configuration language represent integer values, a representation including a decimal point can be used for input: the number is simply truncated towards zero before use. For example, 1.6 would simply represent 1. Because this truncation occurs after the scaling letter, if any, has been applied, the decimal point can be used to express fractions of the scaling value. For example, 1.6M would represent 1677721, which is the truncated integer part of 1.6×1024×1024.

When a constant is used to refer to an amount of memory, it may express a number of words or octets, depending on the processor. For example, on the C6000, such quantities always refer to a number of octets (eight-bit bytes).

```
constant         = decimal constant | hex constant;
hex constant     = "&", hex digits | "0x", hex digits;
hex digits       = hex digit, {hex digit};
hex digit        = digit | "A" | … | "F";
decimal constant = decimal digits,
                   [".", {decimal digit}], [scaling letter];
scaling letter   = "K" | "M";
decimal digits   = decimal digit, {decimal digit};
decimal digit    = "0" |  … | "9";
```

Some examples of numeric constants are given here with their values in decimal.

```
10         10
&10        16
0x12       18
10K     10240
10M  10485760
1.6         1
1.6k     1638
```

String Constants

The only circumstances in which a string constant is required in the configuration language are when an identifier is required that does not obey the configurer’s syntax: operating system file names, for example. Such string constants in the configuration language are simply enclosed in double quotes. No notation is available for including double quotes within the string. The case of the characters in strings is preserved.

The trailing string quote may be omitted if the end of the line terminates the string.

```
string constant = ["""],
                  {? any character except newline and double quote ?},
                  ["""];
```

Some examples of valid string constants are as follows:

```
"string"
"c:\mytasks\x.tsk"
"fred.tsk
```

Identifiers

Each object in the physical system (processors and wires) and in the user's application (tasks and connections) has a unique identifier. This is used by the configurer in error reports, and is also used to specify relationships between the objects. For example, a wire run between links on two named processors.

Identifiers for objects in the configuration language are simply sequences of letters, digits and the special symbols underline "_" and dollar sign "$". The sequence must start with a letter.

```
identifier           =  letter, {identifier character};
identifier character =  letter | digit | "$" | "_";
letter               = "A" | … | "Z";
```

Some examples of valid identifiers follow. Note that the configurer would treat all the last three examples identically, because the case of letters is not significant.

```
proc_5
do$work
root
a_very_long_name
A_Very_Long_Name
A_VERY_LONG_NAME
```

Part of the syntax of each of the configuration language statement types which declare an object is the identifier that is to be used to refer to that object in later statements. For example, the identifier given to a processor is used again in placing tasks on that processor or in wiring the processor's links to those of other processors.

It is sometimes convenient, when an object will not be referred to later, to allow the configurer itself to choose an identifier for an object rather than for the user to invent meaningless identifiers for every object. The declaration statement types all allow a question mark to be used in place of an identifier.

```
new identifier = identifier |  "?";
```

Normally, this special form of identifier is used when declaring wires and connections, as there is at present no statement type that refers back to these objects. Declarations of processors and tasks will almost always require an explicit identifier to be used, as these identifiers are used later when placing the tasks onto the network of processors.

An example of using the question mark form of identifier would be as follows:

```
wire ? root[0] second[3]
```

This statement declares a wire running from link number 0 on processor root to link number 3 on processor

second. The configurer will be able to report errors concerning this wire by reference to the line number and file name of the declaration, but the user will not be able to refer to the wire again.

Statements

Given the definitions of such primitives as numeric constants and identifiers, the high-level syntax of the configuration language can now be presented. The combined input file consists of a number of newline-separated statements, as follows:

```
input file = {[statement], newline};
```

Note that the statement part of the above is optional, allowing for blank lines appearing between statements. This may come about either deliberately, perhaps to improve the readability of the input file, or because the line contained only a comment, which is of course not visible at this level.

Each statement in the input file is one of the following statement types. The different statement types are covered in the subsections that follow.

```
statement = processor statement |
            wire statement      |
            task statement      |
            connect statement   |
            place statement     |
            bind statement      |
            default statement   |
            UPR statement       |
            OPTION statement;
```

There is no restriction on the order in which statements appear in the input file, except that no object may be referred to before it has been declared.

PROCESSOR Statement

```
processor statement   = "PROCESSOR", new identifier,
                         type specification, processor attributes;
processor attributes  = "{" {processor attribute} "}" |
                        {processor attribute};
type specification    = ["TYPE", "="], processor type;
processor attribute   = "CLOCK",    "=", constant              |
                        "CACHE",    "=", cache value           |
                        "KERNEL",   "=", kernel file specifier |
                        "LOAD",     "=", load file specifier   |
                        "BOOT",     "=", boot file specifier   |
                        "AVOID",    "=", avoid spec            |
                        "BUFFERS", "=", constant               |
                        "LINKS",    "=", constant;
processor type        = c6x processor type    |
                        "TMS320C40"  | "C40"  |
                        "TMS320C44"  | "C44"  |
                        "PC";
c6x processor type    = identifier;
cache value           = "ON"         |
                        "OFF"        |
                        constant;
kernel file specifier = string constant;
load file specifier   = string constant;
boot file specifier   = string constant;
physical area         = identifier;
avoid spec            = avoid address, ":", avoid size;
avoid address         = constant;
```

```
avoid size            = constant;
```

The PROCESSOR statement declares a physical processor. If we need to refer to a processor later in the file, it must be declared with a PROCESSOR statement first. Note that you need one PROCESSOR statement for each processor in the system, even if the processors are physically clustered together in some way.

The PROCESSOR (and WIRE) statements must correspond to the real hardware network on which the application will be run. If this is not so, the application will probably fail to load.

TYPE Attribute

The configurer needs to be given information about the type of each processor you will use in an application, and this is provided by a *type attribute* immediately following the processor name in the PROCESSOR statement.

Note

The "TYPE=" string is optional. The following two statements are equivalent:

```
PROCESSOR root TYPE=MyType
PROCESSOR root MyType
```

The type attribute tells the configurer many things, including: the amount and location of the processor's memory, its clock speed, the number of links it has, which kernel file to use, and so on. The valid type identifiers are described in the manufacturer's documentation, but you can get a list of the types acceptable to the configurer using the Diamond utility `ProcType`. The special processor type DEFAULT may be used to select the default processor type. This default will usually have been set by the Diamond installation procedure, but you may set or change it at any time with `ProcType`.

A processor with the name host is assumed to be of type PC. Processors of this type have certain characteristics that we shall discuss later. Despite the name of the type, the host need not be an IBM PC or compatible. Few users will ever need to declare a processor of type PC.

The name "DUMMY" is reserved and may not be used as the name of a processor.

The following example declares a four-processor network. The processors host and other_host are of type PC; the processors root and node are of type C6XBOARD.

```
processor host
processor root       type=c6xboard
processor node       c6xboard
processor other_host type=pc
```

Every processor is assumed to be able to support any user task placed on it by the configuration file. Although certain tasks may not be able to execute on particular types of processor, the configurer cannot check for this and the responsibility for ensuring a valid configuration is the user's.

CACHE Attribute

All C6000 family processors have a two-level cache scheme. This facility is usually (but not always) a net benefit for many applications that are too large to fit entirely into the processor's internal memory. The L1 cache uses a private memory area within the processor and is always enabled. The L2 cache uses some or all of the processor's internal memory to hold cache information. Following reset, the L2 cache is configured to use no internal memory, and so is disabled.

The cache mechanism is always used for internal memory and its operation there is effectively invisible to the running program. Use of the cache (both L1 and L2) with external memory is optional and has potential problems. Cache operation with external memory is not invisible to the running program because the cache is

unable to maintain coherency with DMA operations. A DMA channel can alter the contents of memory that is being cached. Subsequent program accesses will be satisfied from the cache and so get the old memory contents. Coherency must be maintained explicitly in software by the application. This can be expensive and sometimes will reduce performance more than simply disabling the cache. External memory is divided into a number of areas and each area has an associated Memory Attribute Register (MAR). Following reset, all MARs are given values that prevent the cache from operating with the corresponding external memory area. This is particularly important in preventing the cache from interfering with the operation of memory-mapped peripherals. During program execution, preferably during the program's initialisation phase, individual MARs can be set to allow the cache to operate on the selected memory area.

Diamond will automatically enable the cache by setting every MAR that corresponds to external memory defined by the selected processor type.

The CACHE= attribute of the PROCESSOR declaration allows you to control the amount of internal memory used for L2 cache. The default state, with no internal memory being used for cache, is equivalent to specifying the attribute CACHE=OFF. Conversely, CACHE=ON can be used to indicate that the maximum amount of internal memory should be used for L2 cache. For finer control, the CACHE= attribute may be supplied with a numeric argument explicitly giving the amount of internal memory to be used for L2 cache; only certain numeric values are permitted for any given processor type. However, CACHE=OFF is always equivalent to CACHE=0 and CACHE=ON is always equivalent to CACHE=x where x is the maximum permitted value.

You can find out what values are permitted by looking in the documentation for the particular C6000 processor.

LINKS Attribute

This attribute can be used to tell the configurer how many physical links the processor has. There is a default number of links for each processor type. Sometimes the number of links will depend on the way the board or the network is configured, and then the number present will have to be specified with this attribute. Consult the documentation of your board for details.

Traditionally, the links have been for the most part serial link ports. There is no necessity for this to be so, however, and on the C6000, links have been implemented in a variety of ways depending on the architecture of the board; using shared memory, the processor's host port, the two serial ports or additional peripherals, for example. Links implemented in all these different ways, however, will behave in a similar way, and can be handled similarly by the configurers.

CLOCK Attribute

The kernel maintains an internal clock, usually ticking at the rate of 1 tick per millisecond; this clock tick is used in scheduling compute-bound threads and as the timebase for the `<timer.h>` functions. The rate is achieved by configuring one of the C6000's internal timers (TIMER0) with a rate derived from the processor's clock speed. This clock speed is one of the values provided by the processor type. TIMER1 is unused by Diamond.

The configurer has a default setting for each processor type it understands. For processors running at unusual clock speeds, you should specify a CLOCK= attribute in either a PROCESSOR or PROCTYPE declaration so that the kernel clock interrupt rate can be adjusted. For example, the following indicates a processor running at 300MHz rather than 200MHz:

```
processor faster MyProc200 clock=300
```

Note that the argument to the CLOCK= attribute is expressed in megaHertz, not Hertz; CLOCK=300M would be rejected by the configurer.

The following variant is also allowed:

```
processor faster MyProc200 clock=off
```

CLOCK=OFF is equivalent to CLOCK=0. Both forms stop the kernel from using the timer hardware at all. This makes both hardware timers available for you to use directly, rather than TIMER1 only, but disables Diamond's

time-slicing of equal-priority threads and the `<timer.h>` functions. The scheduler will still operate correctly with time-slicing disabled, but threads will always run until they explicitly call a function that may suspend them (e.g., `chan_in_message`).

KERNEL Attribute

Every node of the network must be loaded with a copy of the appropriate version of the microkernel and other necessary software. This is found by opening the correct kernel file. The processor type you give will define the appropriate kernel files for your hardware.

On rare occasions it may be necessary to use a special kernel file for a particular node, rather than the default provided for processors of its type. This may be done with the KERNEL attribute. For example:

```
processor videoboard kernel="video.krn"
```

The configurer looks for the kernel file, whether default or specified by the KERNEL attribute, first in the current directory, and then in each folder specified in the environment variable PATH. This search is not done if the KERNEL attribute specified a full path-name.

Special kernel files will be supplied by 3L or by hardware manufacturers when necessary.

BOOT Attribute

The BOOT attribute allows you to send the uninterpreted contents of a file (a "load object") to a processor or other device. Nothing else will be sent to that processor. In particular, it can have no tasks placed on it. The processor type information must be present, but is ignored; you can specify any existing processor type, even when the "processor" is really a non-processor device.

BOOT is useful in systems with non-processor modules, such as ADCs and DACs, that need to be loaded with configuration information before being used. For example:

```
PROCESSOR root DEFAULT
PROCESSOR adc DEFAULT BOOT="adc.dat"
WIRE ? root[3] adc[1]
```

Note

The load object specified in a BOOT sttribute must be a multiple of 4 bytes long, that is, it must be made up of a whole number of 32-bit words.

AVOID Attribute

The configurer allows areas of external memory to be marked as "do not use". These areas will not be loaded with code or data by the configurer: a common use of this facility is to preserve an area of memory for some non-Diamond use. For example, AVOID= might be used to allocate a memory buffer at a fixed location known by host software or by other processors but not controlled by Diamond at run time.

The format of the AVOID= attribute is as follows:

```
AVOID=base:size
```

For example:

```
processor special MyProc200 avoid=0x04001000:16k
```

BUFFERS Attribute

When a particular processor requires an unusual number of UPR packet buffers, the BUFFERS attribute may be used to override the default setting established by the UPR statement. This only affects speed; the packet routing will always get the data to the correct destination regardless of the number of available buffers.

WIRE Statement

```
wire statement = "WIRE", new identifier, transmitter, receiver,
                         {wire attribute};
transmitter    = link specifier;
receiver       = link specifier;
link specifier = processor identifier, "[", constant, "]" |
                 "DUMMY";
wire attribute = "NOBOOT";
```

The WIRE statement declares a physical connection between links on two different processors.

Caution

The kernels for most processors automatically include a basic set of link drivers, but some types of link need extra modules to be included in the kernel. A reference to a link in a WIRE statement is taken as a request for the configurer to ensure that the relevant driver module is loaded. If you do not reference a link in a WIRE statement, you are unlikely to be able to use that link at all.

Every processor in the network is assumed to be able to control a number of links, numbered upwards starting at 0. In most implementations, the configurer assumes a default number that depends on the processor type or the design of the board. If the PROCESSOR statement includes a LINKS attribute, this overrules the default. The two ends of the wire are each defined by using a link specifier construct.

The position of a link specifier, either transmitter or receiver, will be used to initialise links implemented on hardware that needs to be set into a particular initial state (SDB links, for example). Other than this, because each WIRE statement supports communication in both directions, the two link specifiers in a WIRE statement may usually be interchanged without affecting the statement's meaning. For example, the following statements both [1] declare a wire named yellow_wire running between link 2 of processor proc_one and link 3 of processor proc_two:

```
wire yellow_wire proc_one[2] proc_two[3]
wire yellow_wire proc_two[3] proc_one[2]
```

The PROCESSOR and WIRE statements must correspond to the real hardware network on which the application will be run. If this is not so, the application will probably fail to load.

NOBOOT Attribute

You can use this attribute to prevent a wire from being involved the booting process. The wire will remain available for carrying channel traffic. Your application will fail to build unless all processors can be reached using the remaining wires.

```
wire ? root[2] node[5] NOBOOT
```

DUMMY link specifier

A DUMMY link specifier may be used to declare a wire that is used to communicate with a processor that is not to be involved in the configuration process or an external device. For example,

[1] obviously, putting both these statements into a configuration file would result in an error as the identifier yellow_wire would be declared twice.

```
PROCESSOR root DEFAULT
PROCESSOR node DEFAULT
WIRE ? root[0] node[3] ❶
WIRE ? root[6] DUMMY ❷
WIRE ? DUMMY root[4]
```

❶ Declare a normal wire connection link 0 of root to link 3 of node. If necessary, root[0] will be initialised as a transmitter and node[3] will be initialised as a receiver.
❷ Declare that root's link 6 will be used to communicate with an external device and that it should be initialised as a transmitter.

Using a DUMMY link specifier implies the attribute NOBOOT.

PROCTYPE Statement

The PROCTYPE statement allows you to define custom hardware, and simplifies supporting additional COTS boards. It also allows user-defined processor types to be derived from existing types. The general form of the PROCTYPE statement is as follows:

```
PROCTYPE   new  old   ["{" {attributes} "}", {attributes}]
```

The simplest use of this facility is to define an alias for an existing processor type:

```
proctype  xxx   existing_type
processor root  xxx
processor node1 xxx
processor node2 xxx
```

Changing the single definition of xxx would change the meaning of all subsequent PROCESSOR declarations using the xxx type. PROCTYPE declarations may also introduce new processor types based on existing types but differing in some attributes. For example, this declares a new type like an old type except for processor clock speed:

```
proctype new old clock=300
```

The attributes permitted in a given PROCTYPE declaration differ according to the base processor type. For most C6000 processor types, however, the following attributes are permitted in a PROCTYPE declaration:

Attribute	**Usage**
AVOID=	Prevent Diamond from using a particular block of memory.
CACHE=	Enable or disable the cache. Specify the size of cache to use.
CLEARMEM	Undo the effect of all existing MEM attributes
CLOCK=	Specify CPU clock speed in MHz
FORMAT=	†Internal use only: specify configurer output file format
KERNEL=	Specify Diamond kernel file to be used for this processor
MAP=	†Locate kernel module
MEM=	Declare a block of external memory attached to this processor

All of these attributes, except those marked †, can also be specified in a PROCESSOR statement: if specified both in a PROCTYPE and in a PROCESSOR statement using the processor type, the definition given in the PROCESSOR statement will replace or augment the definition in the PROCTYPE, as appropriate. This is also true when one PROCTYPE is defined in terms of another PROCTYPE.

Kernel Modules

The kernel can be extended when necessary by the inclusion of modules. This is done automatically by the configurer when it detects that a task needs particular facilities. Typically these include link drivers, EDMA handlers, and implementation-specific device drivers.

Each module has a specific numeric identifier, and to gain access to a module, you use the kernel's `SC6xKernel_LocateInterface` function.

SC6xKernel_LocateInterface [Stand-alone]

```
#include <c6kobj.h>
void *SC6xKernel_LocateInterface(void *Kernel, unsigned int ID);
```

This searches the list of modules known to the kernel and returns a handle to the one with identifier value ID. The first parameter should be a pointer to the kernel object that is available to all tasks as `_kernel`. Every call with the same ID will return the same handle.

For example:

```
SC6xExt_Int *xint = SC6xKernel_LocateInterface(_kernel, SIID_SC6xExt_Int);
```

NULL is returned if no suitable module can be located.

MAP= attribute

The configurer allows you to change the particular module that is selected for particular functions. The only time that most users may need to be aware of this mechanism is if they wish to change the kernel's default behaviour, for example, by changing the number of available EDMA channels.

Each MAP= attribute takes the following form:

```
MAP=Logical:Actual
```

"Logical" is general name seen by the configurer when it detects that a module must be included; "Actual" is the specific name of the actual module to be included.

The following example causes the configurer to load a module called "EDMA32" when if is asked to load one called "DMA":

```
MAP=DMA:EDMA32
```

External Memory Specification

MEM Attribute

The MEM= attribute can be used to add an available memory block to an underlying processor type in either a PROCTYPE or PROCESSOR statement. It can be used several times, or in both PROCTYPE and PROCESSOR statements, for a cumulative effect.

Each MEM= attribute takes the following form:

```
MEM=[name:] base:size
```

`name`	is an optional descriptive name for the area, for example SDRAM.
`base`	gives the starting address of the memory area; and
`size`	gives the size of the area in bytes.

The following are valid MEM= attributes:

```
MEM=0x00400000:1M
MEM=SDRAM:0x00400000:1M
```

When the configurer is allocating memory for any purpose, it allocates from the following areas in descending order of preference:

- Internal (on-chip) memory blocks.
- External memory blocks defined as inherent to a built-in processor type.
- External memory blocks defined through MEM= attributes in their order of appearance.

CLEARMEM Attribute

The CLEARMEM attribute can be used in either PROCTYPE or PROCESSOR declarations to cancel the effect of any outstanding MEM= attributes so that accumulation of MEM= information can start afresh. For example:

```
proctype t1    tms320c6201                  !no memory blocks
proctype t2 t1 mem=0x00400000:1M            !1 memory block
proctype t3 t2 mem=0xc0000000:1M clock=300  !2 memory blocks
proctype t4 t3 clearmem mem=0x00400000:1M   !1 memory block, not 3
```

This attribute's main use is in defining large families of related processor types.

TASK Statement

```
task statement       = "TASK", new identifier, task attributes;
task attributes      = "{" {task attribute} "}" | {task attribute};
task attribute       = "INS",          "=", constant            |
                       "OUTS",         "=", constant            |
                       "FILE",         "=", task file specifier |
                       "OPT",          "=", opt area            |
                       "PRIORITY",     "=", constant            |
                       "URGENT"                                 |
                       logical area, "=", memory amount;
opt area             = logical area, {":", location};
logical area         = "CODE" | "STATIC" | "STACK" |
                       "HEAP" | "DATA" | user-defined section;
memory amount        = constant | "?";
task file specifier  = identifier | string constant;
location             = physical area | address;
address              = constant;
user-defined section = string constant;
```

A TASK statement declares a task, and may contain a number of task attribute clauses, each of which describes some aspect of the task. The task's attributes may appear in any order within the statement.

INS Attribute

The INS attribute is used to specify the number of elements in the task's vector of input ports. If the task needs no input ports (because it only requires to send messages to other tasks, never to receive) then this attribute may be omitted or given the value zero.

OUTS Attribute

The OUTS attribute is used to specify the number of elements in the task's vector of output ports. If the task needs no output ports (because it only requires to receive messages from other tasks, never to send) then this attribute may be omitted or given the value zero.

FILE Attribute

> This attribute specifies the task image file where this task is to be found. Task image files are created by the linker. If a TASK statement has no FILE attribute, the configurer assumes that the task image file has the same name as the task, with the appropriate extension. This extension, called the task extension, depends on the processor type. On the C6000, for example, it is ".tsk".

If the FILE attribute is present, its argument is either a string constant, which is the name of the task image file, or an identifier, to which the task extension is added to form the file name. For example, on the C6000, both the following tasks would be loaded from a task image file called myprog.tsk.

```
task abc file=myprog
task def file="myprog.tsk"
```

The configurer looks for the file first in the current directory, and then in every folder specified in the environment variable PATH. This search is not done if the file name is already a full pathname.

Suppose we have the this statement:

```
task this
```

There is no FILE attribute, and assume the PATH variable is set up as follows:

```
► set PATH=c:\mytasks;c:\dos;c:\tasklib
```

The C6000 configurer would search for the task image in the following files, in this order:

```
.\this.tsk
c:\mytasks\this.tsk
c:\dos\this.tsk
c:\tasklib\this.tsk
```

Logical Area Attributes

The configurer must be told the dynamic data storage requirements for each task in an application. This is done using the STACK, HEAP and DATA attributes of the TASK statement. For example:

```
task first  stack=10240 heap=20K
task second data=50K
task third  data=?
task fourth stack=10K    !no heap, not recommended
task fifth  heap=?       !no stack, not recommended
```

In the first example, the STACK and HEAP logical areas are allocated separately, and are given the sizes indicated. In the second example, the stack and heap are allocated to a single area, and are jointly given 50K, while in the third, they are jointly allocated to the largest contiguous area of unallocated memory.

In the fourth and fifth examples, because only one of the areas has been given a size, the other will have no space at all. This will normally cause the task to fail.

CODE and STATIC may not be used as TASK attributes, as the configurer finds the size of these areas by inspecting the task image file.

The argument to one of the memory size attributes is an integer expressing the amount of memory to be allocated to the area in question. Sizes smaller than 128 will not be accepted, to prevent accidental entry of unreasonably small amounts (for example, by typing 1.6 instead of 1.6k). It is also possible to specify "the rest of memory available on the processor" by entering a question mark instead of an integer. On the C6000, which has several distinct areas of memory, "?" is interpreted as "the largest remaining area of suitable memory". Only one task may request this treatment on any particular processor.

Whether these memory sizes are expressed in octets (eight-bit bytes) or in 32-bit words depends on the

processor type. On the C6000, these attributes always specify a number of octets.

OPT Attribute

This attribute of the TASK statement provides a way to give instructions or preferences to the configurer about how you wish memory allocation to be carried out. It has three forms, as shown in these examples:

```
task first  opt=stack opt=seg_red
task second opt=stack:highram
task third  opt=seg_blue:0x28000
```

The first form asks the configurer to place a logical area of the task in the fastest memory possible. The fastest is, of course, the on-chip RAM.

The second asks that a logical section should be placed in a named physical area. Which physical areas you may specify depends on the processor type; for details, you should consult your board documentation.

The third form asks for the user-defined section, seg_blue, to be loaded at the specified physical address. It is extremely rare for this form to be needed; the second form is nearly always adequate.

Warning

Beware of placing sections at address 0. This can result in objects having a zero address which may be interpreted as a NULL pointer. For example, an array of char placed at 0 cannot be output using printf as the argument will be seen as a NULL pointer and interpreted as an empty string.

Using either of the first two forms of this attribute does not guarantee that what you have requested will in fact happen. The OPT attributes of all the various tasks on a processor, the sizes of their logical areas and the sizes of the physical areas available are all taken into account by the configurer when it performs the memory allocation.

As well as the standard logical area names, CODE, DATA, STACK, HEAP and STATIC, you can specify named sections of the task, defined by the programmer. Where such sections have names that do not conform to the configurer’s conventions for identifiers, you must enclose the name in quotes:

```
Task fourth opt= ".tables"
```

PRIORITY Attribute

This attribute provides a way to specify the priority level at which the task's initial thread (`main`) is to be started. The range of priorities allowed varies between processors; on the C6000, for example, priorities in the range 0–7 may be specified; level 0 has the highest priority and level 7 the lowest. For example:

```
task abc priority=5
```

If no PRIORITY attribute is given, the task's initial thread is started at priority level 1. A task may also be started at priority 0 by giving the URGENT attribute (see below).

URGENT Attribute

This attribute specifies that the task's initial thread is to be started at the urgent priority level, that is, at priority level 0. The two following examples have exactly the same effect:

```
task abc urgent
task abc priority=0
```

Port Specifiers

After the declaration of a task, its ports may be referred to in much the same way as the links of a processor, by a port specifier construct consisting of the task identifier followed by a number enclosed in square brackets:

```
port specifier = task identifier, "[", constant, "]";
```

For example, either input or output port number 5 on task user would be specified as user[5].

Note that a port specifier as given here does not indicate whether the port concerned is an input port or an output port, that is, whether the index given is into the task's vector of input ports or into its vector of output ports. This information is provided by the context in which the port specifier appears. In the CONNECT statement, the port specifier's direction is determined by its position within the line. In the BIND statement, the port specifier is preceded by a direction word (INPUT or OUTPUT).

CONNECT Statement

```
connect statement     = "CONNECT", new identifier,
                        output port specifier, input port specifier,
                        {connect attribute};
output port specifier = port specifier;
input port specifier  = port specifier;
connect attribute     = connection type | "SHORT", "=", constant;
connection type       =  "PHYSICAL" | "VIRTUAL";
```

The CONNECT statement connects an output port on one task with an input port on another task. For example:

```
connect ? driver[0] upc[0]
! connect output port 0 of driver to input port 0 of upc
connect ? upc[0] driver[0]
! connect output port 0 of upc to input port 0 of driver
```

While the WIRE statement describes a hardware connection in both directions between two processors, the CONNECT statement describes a logical, uni-directional connection from one task to another. The connection creates two new channels, an input channel and an output channel. The output channel is bound to the output port of one task, and the input channel is bound to the input port of the other task. The configurer arranges for output sent to the output channel to be received by the input channel. Communication in both directions between a pair of tasks therefore requires two CONNECT statements, as in the example above.

The tasks being connected need not be on the same processor. In some implementations, the CONNECT statement will be supported by sophisticated run-time software which will arrange for messages to be forwarded to the appropriate task, wherever it is. In others, the CONNECT statement will be mapped directly onto the hardware link connections, and as a result, there may be restrictions on the placement of tasks which are joined in this way.

These connections may be named. This allows you to give a name to the input or output channel that the named connection has created for the task. You can then access the channel directly using its name. This is described under `INPUT_PORT` and `OUTPUT_PORT`. If you do not name the connections, or do not wish to name the channels, you access the channels by indexing into the vectors of ports passed to the task as arguments to `main`.

A number of optional connection attributes may follow the input port specifier in a CONNECT statement, in any order.

Connection Type Attribute

A connection may be explicitly specified as either VIRTUAL or PHYSICAL, but not both. For example:

```
connect ? sender[2]      receiver[0]  virtual
connect ? AtoD_driver[0] FIRfilter[0] physical
```

Physical connections are directly mapped onto WIREs. Virtual connections are more flexible. See here for more about the differences between virtual and physical connections. Some implementations only support PHYSICAL connections.

If neither VIRTUAL nor PHYSICAL is specified, the configurer will use a default setting, which is determined as follows. Connections are normally virtual by default. This behaviour can be changed using the DEFAULT CONNECT statement. The configurer command-line switch "–p" can be used to make all connections physical by default. The configurer will report any clashes between these different ways of specifying the default type of a connection.

Sometimes the configurer will not be able to make a connection virtual. This can happen if either of the following conditions is true:

- One of the tasks to be connected is placed on a processor type (transputer or Alpha) that does not presently support virtual connections.
- One of the tasks to be connected was built using a version of Diamond for the C6000 prior to V2.1, and therefore does not support virtual connections.

If this happens, the connection will fall back to being physical. If a virtual connection was explicitly requested (using the VIRTUAL attribute), the configurer will give a warning message. If there are not enough spare WIREs in the configuration to accommodate an extra physical connection, the configurer will give a further error message.

SHORT Attribute

The Virtual Channel Router (VCR) system task keeps a small buffer at the receiving end of each virtual connection. Short messages that fit entirely within this buffer are handled more efficiently than longer messages. If you do not specify otherwise, each buffer will be 64 octets long. This default setting may be overridden as follows:

```
connect ? task1[0] task2[1] short=128
```

This connection will use a 128-octet buffer. SHORT=0 is allowed, and forces all messages sent over the connection to use the slower long-message mechanism. The value given for SHORT must be less than the maximum UPR packet size.

The SHORT attribute only makes sense for virtual connections: it is ignored for physical ones.

PLACE Statement

```
place statement       = "PLACE", placing;
placing               = task placement | connection placement;
task placement        = task identifier, processor identifier;
connection placement  = connection identifier, wire identifier;
processor identifier  = identifier;
task identifier       = identifier;
connection identifier = identifier;
wire identifier       = identifier;
```

The PLACE statement determines on which processor a particular task will execute, or which wire a connection will use. Every task must be placed on some processor, but placement of connections on wires is optional—the configurer can automatically choose a suitable wire for you. Except for the restrictions of space, there is no limit on the number of tasks that may be placed on a single processor.

Examples of the use of this statement might be as follows:

```
processor root
processor node

wire fast root[0] node[3]
wire slow root[1] node[4]
```

```
task one ins=1 outs=1
task two ins=1 outs=1

connect f1 one[0] two[0]
connect f2 tow[0] one[0]

place one root
place two node
place f1  fast
place f2  slow
```

BIND Statement

```
bind statement = "BIND", binding type, port specifier, binding value;
binding type   = "INPUT"} | "OUTPUT";
binding value  = "VALUE", "=", constant;
```

Warning

Normally, ports are only bound by means of the CONNECT statement; ports left unbound are pointed at unique dummy channels so that attempts to send or receive messages through them cause the minimum of harm; the thread causing the attempt to communicate over the unbound port simply pauses indefinitely rather than causing failure of possibly all threads running on the processor.

The BIND statement allows you to set the contents of a port explicitly to some literal value that does not correspond to a channel.

One application of the BIND statement is to pass an integer parameter to a user task. For example, suppose the configuration file contained the following statement:

```
bind input event_handler[5] value=17
```

This value (17) could be accessed within the task event_handler by code like the following:

```
#include <chan.h>

main (int argc, char *argv[], char *envp[],
      CHAN * in_ports[], int ins,
      CHAN *out_ports[], int outs)
{
   int parameter;
   parameter = (int)in_ports[5];
   ...
}
```

This technique can be used to allow several otherwise identical tasks to behave differently. For example, tasks executing on a fast processor can have this fact indicated to them by means of a parameter value, and use a more processing-intensive algorithm for the solution of some problem. Another use of this parameter facility is to "label" each task with a unique identifier.

Warning

If an arbitrary value is supplied for a port binding and an attempt is then made to send or receive a message using that port, the processor on which the task resides will most probably crash.

DEFAULT Statement

```
default statement = "DEFAULT", "CONNECT", connection type;
```

There are only two forms of the DEFAULT statement:

```
default connect physical
default connect virtual
```

Only one of these statements may appear in the input to the configurer. The DEFAULT CONNECT statement governs how the configurer treats any CONNECT statements that do not explicitly specify whether a virtual or physical connection is required. See the description of how defaults are handled, in "Connection Type Attribute" Note that the effect of DEFAULT CONNECT is global. All CONNECT statements are affected, even those that come before the DEFAULT statement in the input file.

UPR Statement

```
UPR statement = "UPR", {UPR attribute};
UPR attribute = "MAX",     "=", constant |
                "BUFFERS", "=", constant;
```

The UPR statement allows you to tune various settings in the Universal Packet Router (UPR) system task.

MAX Attribute

The MAX attribute specifies the maximum size of UPR network packets, in octets. The default is 2060 octets. Increasing this value will speed up UPR communications if fewer UPR packets are then required to transmit a message of a given size. On the other hand, decreasing MAX reduces the amount of memory required for UPR packet buffers.

MAX must be an exact multiple of 4 between 128 and 65532, inclusive.

BUFFERS Attribute

UPR runs a number of virtual links over each physical link under its control. The number of virtual links on each physical link i calculated by the configurer and depends only on the network topology: it is not directly related to the number of virtual connections routed over that link. For each virtual link, UPR allocates a number of buffers; the default is two. Increasing the number of buffers on a virtual link may increase the performance of communications over that virtual link, particularly for long messages routed through intermediate nodes.

The BUFFERS attribute of the UPR statement can be used to change the number of buffers that will be allocated to each virtual link. The new setting applies to all processors, except where the BUFFERS attribute of an individual PROCESSOR statement is used to override it. The number of buffers specified must be in the range 1 to 50, inclusive. The size of each buffer is controlled by the MAX attribute of the UPR statement; see above.

OPTION Statement

```
OPTION statement = "OPTION", {OPTION selection};
OPTION           = "LoadCheck"  |
                   "NoLoadCheck";
```

LoadCheck option

```
OPTION Loadcheck
```

The LoadCheck option instructs the configurer to generate extra information in the application file to check that

the actual network used to run an application matches the network specified in the configuration file. The checks only test the types of the processors and the links being used for loading. Checking needs to be enabled by the loading software. Host programs that know nothing of this checking can continue to load applications by sending the contents of the application file down the host link without interpretation; no checks will be performed.

NoLoadCheck option

```
OPTION NoLoadCheck
```

The NoLoadCheck option prevents the configurer from placing checking information in the application file. This is the default option.

Chapter 8. The Configurer

This chapter describes the version of the configurer, `config.exe`, which is distributed with the C6000 edition of Diamond. The configurer implements the configuration language described in Configuration Language Reference.

The configurer can build applications to run on C6000 networks, on C4x networks, and on networks containing a combination of these processors. The way it handles the processors of a network, and the tasks which are placed on them, depends on the type of each processor.

For the most part, this chapter fills in the details that are related especially to the C6000. For details of the way in which other processors are handled, you should consult the appropriate User Guide.

Using the Configurer

Invoking

The configurer is invoked with a command of this format:

```
config switches input-files output-file   or
config switches focus
```

switches control the configurer's options. Although shown here in a fixed position they may appear anywhere within the command.

input-files are one or more files containing configuration language statements.

output-file gives the name of the application file to be created. The conventional extension is ".app", but the whole name must be supplied.

focus identifies both the input configuration file and the application file to be created. The configuration file will be `focus.cfg` and the application file `focus`. For example, the following two commands will have identical effects:

```
► config test
► config test.cfg test.app
```

You may also invoke the configurer without any parameters, when it will display its identity and stop.

Switches

Switches may be placed anywhere in the command; they should be introduced with a "-" character. [1] For example:

```
config -v test.cfg test.app -L
```

If two switches are placed together, there must be a space between them:

```
► config -L -v test.cfg test.app
```

The configurer recognises the following switches:

-A	Prevent the network loader from sending task information or ready-to-go requests to the host. This is intended for use when no host server may be available to service these requests, e.g., in a standalone ROM-based system.

[1] For compatibility, switches may also start with "/"

-C	Exhaustively check the tables used by the communication software, ensuring that they are correct, and proving that deadlocks are impossible.
-G	Generate symbol table files for all of the tasks in the system. The name of each symbol table file will be the name of the task with ".out" appended. Additional symbol table files will be created for the kernel and other system tasks.
-L	Output information to stdout about memory addresses allocated by the configurer. Compare this with the output generated by `-l`. The listing shows: • the memory addresses allocated for the kernel on each processor; • the addresses allocated to each section of every task; • the external symbols defined by each task; • the address allocated to each external symbol.
-l	Create a brief listing file giving information about memory addresses allocated by the configurer. Compare this with the output generated by `-L`. The listing shows: • the memory addresses allocated for the kernel on each processor; • the addresses allocated to each section of every task; Use -L if you want the listing to include external symbols.
-M	Create a map relating PROCESSOR names in the input configuration file to the UPR node numbers used by the application at run time. The map is written to the standard output stream
-P	Make all CONNECT statements PHYSICAL by default. If -P is not used, connections are virtual by default.
-ROM	This option is no longer needed with Sundance applications.
-V	Display a "running commentary" during configuration

Input Files

The input files should be one or more configuration files. These contain instructions for building the application, written in the language defined in Configuration Language Reference.

Use of Files

The default extension for the filenames of task image files is ".tsk". This means that the following will each result in file abc.tsk being opened:

```
task abc
task xxx file=abc
task yyy file="abc.tsk"
```

Processor Types

Each processor you wish to use must be qualified by a TYPE=name attribute. This attribute must immediately follow the processor name. It not only indicates that the processor is a C6000, but also identifies the board or module type. From this the configurer knows the physical memory areas that are present and their sizes, the number and nature of the links, and so on. You should consult the documentation for your C6000 board to find out the correct TYPE attribute to use for your processors. The "TYPE=" text is optional:

```
PROCESSOR Capture TYPE=XYZ123
PROCESSOR Display XYZ123        ! Interpreted as TYPE=XYZ123
```

You may use the special type name DEFAULT to refer to the processor type set as default by the ProcType

utility.

Memory Use

Memory Divisions

Logical Memory Areas

The memory used by a C program is divided into a number of logical memory areas.

Code storage	is used to hold the executable instructions of the program itself, together with some constant data and control information
Static storage	is used to hold static and external variables, including variables declared at the global level.
Stack storage	(sometimes referred to as workspace) is used for auto variables. The stack is also used for function calls and passing parameters.
Heap storage	is used to hold all variables created by calls on malloc, etc. The run-time library also uses the heap internally for I/O buffers and other things.
User-defined sections	are created and managed by the user. They can be defined in a C source program by using the CODE_SECTION and DATA_SECTION pragmas; see section 7.6 of the Optimizing C Compiler.

These areas are mapped onto an arrangement of physical memory that varies widely between different C6000 boards and environments. The configurer knows the arrangement of physical areas from the TYPE attribute of the PROCESSOR statement.

Physical memory Areas

On-chip memory	The C6000 processors include areas of on-chip, or internal, memory. The size and properties of these areas depend on the particular processor in use. Some or all of this internal memory can be used as cache.
External memory	C6000 modules may have one or more areas of external memory, depending on the design of the board. These may be of different speeds.

Memory Mapping

Because of the wide variation in memory hardware, different C6000 modules and environments may need different methods of mapping the logical areas of a C program into physical memory. This mapping is done at configuration time by the configurer. The decisions on how the mapping is to be done are made using the following information:

- The design of the board, including such information as the speed and size of the various physical memory blocks, so far as these can be known;
- The sizes of the logical areas (including user-defined sections) of all the tasks making up the application;
- Hints from the user about which logical areas of which tasks it would be profitable to optimise, that is, place in fast memory;
- Explicit requests from the user to place certain logical areas in specific physical areas, or at specific memory addresses.

On each processor, the configurer will map the logical areas into physical memory in the following order:

1. logical areas given explicit addresses;
2. logical areas appearing in OPT statements;
3. the interrupt vector
4. the kernel and other system tasks;
5. remaining logical areas with known sizes;
6. data areas created by the configurer for task management;
7. a single STACK, HEAP or DATA area without an explicit size.

The configurer will always use fast memory in preference to slow. This means using the internal (on-chip) memory if it is available, and failing that the fastest available area of external memory.

The configurer is very good at allocating memory and its choices are usually good enough for most applications. You can see how the configurer has allocated memory to your tasks by examining the listing file which you can generate with the -L command switch.

The next two sections describe how the user can adjust the memory mapping to improve performance. See also the description of the TASK statement in Configuration Language Reference.

The OPT Attribute

The OPT attribute of the TASK statement is used to indicate to the configurer that you wish a logical area to be placed, if possible, in fast memory. The attribute has the following format:

```
opt=logical-area
```

The logical-area could be CODE, STATIC, STACK, HEAP or DATA, or the name of a user-defined section:

```
task integral stack=23K opt=stack opt=my-segment
```

Here, my-segment is a user-defined section; it may have been defined using a CODE_SECTION pragma, for example.

The logical area name DATA may be used to refer to the heap and the stack, treated as a single, combined area; see below.

The fact that you code some OPT attributes does not necessarily mean that the mapping you want will happen:

- There may not be enough fast memory available.
- The configurer does not guarantee that the requirements of the various tasks will be dealt with in any particular order. By the time any particular task is dealt with, other tasks may have used up the resources it is requesting.
- The configurer has a fixed idea about the order in which multiple OPT attributes on one TASK statement should be processed.
- The configurer cannot optimise a logical area whose size has not yet been determined; see the following section.

Logical Area Sizes

The configurer must know the sizes of the logical areas of all the tasks on a processor. This is easy for code and static storage, as the linker stores the information in the task image files, but you must explicitly supply sizes of the stack and heap areas.

Working out the stack and heap requirements of a task can be quite difficult. Unfortunately, the compiler cannot do it, as stack and heap use depend upon the program's requirements at run time.

For the size of the stack, you must work out how much space is needed for all the functions that may be active at once, based on the sizes of the auto variables they use. Each level of function calling uses a minimum of about three words of stack space in addition to the space required for function data. Also, library functions use varying amounts of stack space as working storage.

Similarly, for the heap, you must estimate the maximum needed at any one time. Heap storage is currently allocated by the run-time library in blocks of 4K octets (eight-bit bytes), so if your task uses the heap, be sure to allocate at least that much space for it.

In addition to the amount of space you estimate your task actually needs, you must allocate an extra 200 octets (330 octets for processors from the C64 family) on the stack. The kernel uses this to manage the task.

The absolute minimum amount of space you need to execute the simplest of tasks safely is given by the macro THREAD_MIN_STACK.

Bear in mind that if a task exceeds its stated memory requirements the whole system will probably crash, so err on the side of caution. It is a good idea to add at least 1K or 2K octets of extra stack overflow space unless you are absolutely sure the task will never require more space than you have calculated.

For tasks that use the heap from a combined stack and heap area (DATA=), a good rule of thumb would be to allocate at least an extra 8–10K octets.

Once you have decided the sizes of the stack and heap areas, you can chose between two ways of specifying them to the configurer. These can, if you wish, be used in conjunction with the OPT attribute discussed above, or by themselves.

DATA attribute

You can specify the stack and heap requirements of tasks together, using the DATA attribute. This will assign a single block of memory to the stack and heap jointly and it will be divided between them dynamically at run time, as required. For example:

```
task integ data=20k
```

It is also possible to use a "?" with the DATA attribute, like this:

```
task integ data=?
```

This allocates the largest contiguous block of memory that is not explicitly assigned to logical areas of this or any other task. Only one logical area attribute on a processor may be specified in this way. It is also possible to omit all memory specifications from a task. Although the configurer will issue a warning if you do this, it will treat it as equivalent to coding DATA=?.

Separate Stack and Heap

Rather than using DATA=, you can specify the requirements of the stack and heap separately. This enables the configurer to map them to different physical memory blocks, if this would be useful. For example:

```
task integ stack=5k heap=15k
```

It is possible to use a "?" with the STACK or HEAP attributes, like this:

```
task integ stack=5k heap=?
```

This will allocate the largest block of contiguous unassigned memory on the processor to the logical area specified. Once again, only one logical area attribute on a processor may be specified in this way.

Warning

If you specify one of these logical areas without specifying the other, the unspecified area is given no memory at all. This will nearly always result in the application failing.

Explicit Placement of Logical Areas

The OPT attribute can be used to place a logical area (including a user-defined section) in a specific physical area. For example:

```
task one opt=stack:highram
task two opt=my_seg:lowram
```

The first of these would place the STACK area of task one in the highram physical area. The second would place the my_seg user-define section of task two in the lowram physical area. The processor on which the task is eventually placed must have highram and lowram memory areas; these would normally be defined for you in the processor's type definition.

In rare and extreme cases, you can place a logical area (including a user-defined section) at an explicit address. For example:

```
opt=thing:0x1000
```

This would result in the "thing" section of the task being located at absolute address 1000_{16}.

Building a Network

Restrictions on Network Configuration

The following restrictions currently apply to the target network; you only have to consider them if you make use of physical channels.

- As we have seen, the WIRE statement describes the hardware links between processors. Every processor in a network must be reachable from the root via a sequence of WIREs. This path may pass through any number of intermediate nodes. This condition ensures that the network can be booted through its links. (There may be more than one possible path to any given node.)
- The Universal Packet Router (UPR) system task will be present on every processor containing tasks that use virtual channel communication or are linked with the full run-time library. Processors on which UPR is present are called UPR nodes. Every UPR node must be reachable from every other UPR node by some sequence of WIREs that only passes through UPR nodes. This condition prevents the construction of networks with dangling UPR nodes, cut off from the rest by intermediate processors that cannot forward messages because they are not running UPR. Most networks that use virtual channel communications will automatically meet this condition, because every node will be a UPR node. The configurer will report a "UPR connectivity failure" if the rule is broken. This may happen if an intervening node has no tasks placed on it, or contains only stand-alone tasks that do not use virtual channels (in particular, no executable

references to the channel input and output functions). The configurer will attempt to place a suitable dummy task on otherwise empty processors to prevent the failure.

- The Global File Services (GFS) system task will be present on every node that contains a task linked with the full run-time library. A path of UPR nodes must exist from the root processor to every node on which GFS is present, and the processors on this path must all be of the same family (all C6000 or all C4x). This condition ensures that the GFS system tasks, which support the standard I/O facilities of the full run-time library, can always communicate with the root processor, and therefore with the server. Again, most networks using virtual channels will satisfy this condition automatically.

Restrictions on Physical Channels

Additional network configuration restrictions apply when physical channels are used:

- If two tasks are to be connected by a physical channel, there must be at least one WIRE running directly between the processors on which the tasks are placed. Intermediate nodes cannot forward messages sent on physical channels.
- Each link (WIRE) can carry only two physical channels, one in each direction. If more physical channels are required than there are links available, the configurer will report an error. Note that any virtual channels using a WIRE will consume both of the two physical channels available on that WIRE.
- Any WIRE that is allocated to a physical channel is not available for use by UPR (see the previous restriction). This may result in not enough spare WIREs being available to connect all the UPR nodes. If that happens, the configurer will report a "UPR connectivity failure". For example, consider a network of only two processors, in which a physical channel connects one task on the root processor to another task on the second processor. This can be done with just one WIRE. However, if any virtual channel connections are also required between the two processors, or any task on the second processor is linked with the full run-time library and requires GFS, then two WIREs will be needed: one for the physical channel and one for UPR. If a virtual channel were used instead of a physical one then a single WIRE would be sufficient.

Messages

In this section, all the messages output by the current version of the configurer are listed, with a brief description of what they mean. The messages are arranged in alphabetical order, except that symbols like *filename* are not included. So

```
ERROR: name is not a task
```

comes before

```
ERROR: KERNEL attribute incompatible with other attributes
```

After the configurer outputs an error message, it will usually try to carry on as far as it can. When it can go no further, the following message will be output:

```
PROCESSING ABANDONED
```

Any error messages issued by the configurer but not listed here are usually the result of using corrupt task image files. If the cause of the error is not clear, please contact 3L.

ERROR: AVOID=addr:size fails: "no such memory on processor proc"

ERROR: bad magic number in COFF file, hexnum (expected hexnum or hexnum)

ERROR: bad magic number on optional header

An input task file is badly formatted, and seems to be corrupt.

ERROR: BOOT attribute incompatible with other attributes

The BOOT attribute of the PROCESSOR statement may not be used with the KERNEL or LOAD attributes; nor may there be more than one BOOT attribute in one statement.

ERROR: cannot access processor kernel file name

The configurer cannot find the file containing the named kernel file. Check that your search path references the Diamond installation folder.

ERROR: cannot allocate x bytes for section name of task name

There is no area of consecutive bytes large enough to hold the named section.

ERROR: cannot allocate x bytes of private space

The configurer communicates information about the application to the kernel in a small area of memory. This error indicates that not enough continuous memory is available for this area.

ERROR: cannot boot a network which has no root

The network contains no host or root processor.

ERROR: cannot place section name from task name at address: memory not available

An OPT attribute has been used to place the named section at the given address. The error indicates that part of the memory required either foes not exist or has already been allocated to another section.

ERROR: cannot find a link on which to place connection: name

Each WIRE statement is able to support two CONNECT statements, one in each direction. This error message indicates that there was no WIRE statement left to support the CONNECT statement with the specified name.

ERROR: cannot find load object filename

The file named in a LOAD, KERNEL or BOOT attribute of a PROCESSOR statement could not be found. Check whether the file exists; also check the search path.

ERROR: cannot open application output file filename

The configurer could not open the application file you asked it to create.

ERROR: cannot open input file filename

The configurer could not open the specified configuration file.

ERROR: cannot open task image file filename for task name

The configurer could not open the task image file filename. It is needed to build task name.

ERROR: cannot reach processor name from the host

Every processor in the network must be connected, directly or indirectly, with the root processor, which is connected to the host; detached sections of the network are not allowed. The connections are made with WIRE statements, but WIRE statements marked NOBOOT are not considered when determining a path from the host.

Note

Some processors may provide links that are unable to participate in the loading process and so the NOBOOT attribute will be assumed for any WIRE statements referencing them.

ERROR: cannot read load object from file

There was an error when attempting to read data from the file specified in a LOAD, KERNEL or BOOT attribute of a PROCESSOR statement.

ERROR: connection name is already connected

The specified name has already been used in a CONNECT statement.

ERROR: connection c-name has already been placed on wire w-name

A PLACE statement has attempted to place the connection c-name onto a wire when it has already been placed on the wire w-name.

ERROR: could not close application output file

There was an error when the configurer tried to close the output application file.

ERROR: error positioning file

There was an error while the configurer was locating information in a task image file. This usually indicates that the task image file is corrupt.

ERROR: error while reading load object filename

There was an error when attempting to read data from the file specified in a LOAD, KERNEL or BOOT attribute of a PROCESSOR statement.

ERROR: expected a 'character'

This error message is given when the configurer was expecting a particular character, such as '=', in its configuration file, but found something else.

ERROR: expected a bootfile name

The configurer expected to find a filename in the BOOT= attribute, but did not.

ERROR: expected a file name for FILE attribute

The configurer expected to find a filename in the FILE= attribute, but did not.

ERROR: expected a kernel file name

The configurer expected to find a filename in the KERNEL= attribute, but did not.

ERROR: expected a keyword at the start of the line

What the configurer found could not be the name of a statement.

ERROR: expected a loadfile name

The configurer expected to find a filename in the LOAD= attribute, but did not.

ERROR: expected a physical area name

What was found was not a valid physical area name for this processor type.

ERROR: expected a processor attribute keyword

On the C6000, the allowed attributes of the PROCESSOR statement are currently TYPE, LINKS and KERNEL.

ERROR: expected a processor type keyword

This refers to the TYPE= attribute of the PROCESSOR statement. See your hardware documentation to find out what type names are allowed.

ERROR: expected a task attribute keyword

The allowed attributes for the TASK statement are INS, OUTS, FILE, OPT, PRIORITY and

URGENT. You may also use one the following logical area names as an attribute: DATA, STACK and HEAP.

ERROR: expected an area name to avoid

The AVOID= attribute of the PROCESSOR statement must be followed by the name of a physical memory area.

ERROR: expected an area name to optimise

The OPT= attribute of the TASK statement should be followed by the name of a logical area. The following are allowed: DATA, STACK, HEAP, CODE, STATIC, or the name of a section in the task image file.

ERROR: expected an identifier here

An identifier is a sequence of letters, digits and the symbols "_" and "$". It must start with a letter.

ERROR: expected an integer constant

See section "Numeric Constants". Briefly, a decimal integer constant is a sequence of digits. Hexadecimal constants start with "0x". A constant may end with a scaling letter, "K" or "M", indicating that the number is to be multiplied by 1024 or 1024×1024 respectively.

ERROR: expected INPUT or OUTPUT keyword

The BIND statement keyword must be followed immediately by one of the words INPUT or OUTPUT.

ERROR: expected VALUE=

The BIND statement must end with a VALUE= attribute.

ERROR: extra stuff found at end of line

There are extra characters at the end of the line that cannot be understood as part of the statement.

ERROR: failed to write number bytes to output file

There was an error while writing data to the output application file.

ERROR: file name has already been specified

There should not be more than one FILE attribute on a TASK statement.

ERROR: flood configuration not supported for processor type type

The present version of the C6000 configurer does not support flood configuration.

ERROR: name has already been declared as object

The identifier name has already been used in a PROCESSOR, WIRE, TASK or CONNECT statement.

ERROR: name has not been declared

You have used the identifier name, but it has not so far been declared.

ERROR: impossible number of symbol table entries

An input task file is badly formatted, and seems to be corrupt.

ERROR: incompatible memory allocation for task name

The logical memory area DATA includes both STACK and HEAP. For this reason, if you specify DATA, you cannot specify STACK or HEAP as well.

ERROR: INPUT port task[number] has already been bound

You cannot do more than one BIND statement on a port.

ERROR: INPUT port task[number] has already been connected

You cannot BIND a port that has already been used in a CONNECT statement.

ERROR: input port task[number] is already connected

You cannot CONNECT an input port which has already been used in another CONNECT statement.

ERROR: insufficient memory to allocate another number bytes

The configurer itself does not have enough memory to configure the application.

ERROR: insufficient memory to extend a block to number bytes

The configurer itself does not have enough memory to configure the application.

ERROR: number is an unreasonably small memory size

The minimum memory size that can be specified in any statement is 128 bytes.

ERROR: name is not a processor

The configurer expects the name of a processor, as defined by a PROCESSOR statement.

ERROR: name is not a task

The configurer expects the name of a task, as defined by a TASK statement.

ERROR: name is not a valid area name to be avoided

The AVOID= attribute of the PROCESSOR statement should be followed by the name of a physical area.

ERROR: name is not a valid area name to be optimised

The OPT= attribute of the TASK statement should be followed by the name of a logical area. The following are allowed: DATA, STACK, HEAP, CODE, STATIC, or the name of a section in the task image file.

ERROR: character is not a valid hex digit

The valid hex digits are the decimal digits, 0–9, and the letters A–F (or a–f).

ERROR: number is not a valid input port number for task name

The input ports for a task are defined by the INS= attribute of its TASK statement. If, for example, you write INS=5, then input ports 0–4 are valid.

ERROR: number is not a valid link number for processor name

The links on a C6000 that can be used by the configurer are limited to the number defined in the LINKS attribute of the PROCESSOR statement.

ERROR: number is not a valid output port number for task name

The output ports for a task are defined by the OUTS= attribute of its TASK statement. If, for example, you write OUTS=5, then output ports 0–4 are valid.

ERROR: name is not a valid physical area name

The physical area name supplied was not one of those allowed for this processor type.

ERROR: number is not a valid port number

Negative port numbers are not allowed.

ERROR: name is not a valid processor attribute keyword

On the C6000, the allowed attributes of the PROCESSOR statement are currently TYPE, LINKS and KERNEL.

ERROR: name is not a valid processor type

This refers to the TYPE= attribute of the PROCESSOR statement. See the documentation for your hardware to find out what type names are allowed.

ERROR: name is not a valid statement type keyword

The given name is not one of the keywords that may start a statement. See the list of allowed statement types.

ERROR: name is not a valid task attribute keyword

The allowed attributes for the TASK statement are INS, OUTS, FILE, OPT, PRIORITY and URGENT. You may also use one the following logical area names as an attribute: DATA, STACK and HEAP.

ERROR: KERNEL attribute incompatible with other attributes

There is more than one KERNEL attribute in this PROCESSOR statement.

ERROR: LINKS attribute specified more than once

Only one LINKS attribute is allowed on a PROCESSOR statement.

ERROR: multiple UPR BUFFERS definitions

You may only specify UPR BUFFERS=n once in your configuration file. That setting applies, by default, to all processors. If you want a different setting for some processors, use the BUFFERS attribute of the PROCESSOR statement.

ERROR: multiple UPR MAX definitions

The maximum UPR packet size is a global constant throughout an application. Therefore you may only specify UPR MAX=n once in your configuration file.

ERROR: no input files have been specified

At least one input file is needed.

ERROR: no output file has been specified

The name of an output application file must be supplied on the command line.

ERROR: OUTPUT port task[number] has already been bound

You cannot do more than one BIND statement on a port.

ERROR: OUTPUT port task[number] has already been connected

You cannot BIND a port that has already been used in a CONNECT statement.

ERROR: output port task[number] is already connected

You cannot CONNECT a port that has already been used in another CONNECT statement.

ERROR: priority value must be between 0 and 7

7 is currently the maximum value allowed for the PRIORITY attribute.

ERROR: processor type has already been specified

Only one TYPE attribute may be given in a PROCESSOR statement.

ERROR: relocation information has been stripped

An input task file is badly formatted, and seems to be corrupt.

ERROR: task name cannot be given the rest of memory on processor name, as this has already been given to task name

By using a TASK attribute of the form logical-area=?, you can assign to that area all the available space on the processor, but this can only be done once on each processor.

ERROR: task name has already been placed on processor name

Another PLACE statement has already been given for this task.

ERROR: task name has no entry point

The configurer cannot locate the address of the first instruction of the named task.

ERROR: task name has not been placed on a processor

There was no PLACE statement for this task.

ERROR: task image file for task name is corrupt

Possibly it is not a task image file at all; or perhaps there has been some disruption of the host file system.

ERROR: task image file is not executable

An input task image file cannot be executed. The most usual cause of this is that the linker terminated in error; for example, because an external reference could not be satisfied.

ERROR: the PLACE statement cannot be used on name (a sort)

You have tried to PLACE a thing of type sort called name. You may only place tasks on processors or connections on wires. A common cause of this error is to invert the arguments to PLACE and attempt to place a processor on a task or a wire on a connection.

ERROR: these links are already connected to another wire

At least one of the two links mentioned in a WIRE statement has already been used in another WIRE statement.

ERROR: this configurer restricted to one processor

This message is output by the version of the configurer that is distributed with single-processor versions of Diamond. It indicates that the user has attempted to define a processor network with more than one processor.

ERROR: unable to create loader with special RAM attributes

The RAM attribute, which is accepted by the configurer for some other processors, is not allowed with the C6000.

ERROR: unexpected size for optional COFF header

An input task file is badly formatted, and seems to be corrupt. This is usually the result of referencing a file that does not contain a Diamond task, for example:

```
TASK strange FILE= "thing.c"
```

ERROR: unknown option string

This string, which appeared on the configurer's command line, is not recognised as a valid option.

ERROR: UPR connectivity failure: number processors could not be reached

There must be a copy of UPR on every node between the root and any non-root processor that requires host services (such as C standard I/O). Similarly, for one node to communicate successfully with another via a virtual channel there must be a copy of UPR on every processor between those two nodes (see sections "Restrictions on Network Configuration" and "Restrictions on Physical Channels". One way to make sure UPR gets loaded onto a particular node is to link a task on that processor with the full run-time library. The configurer does not automatically load UPR onto "empty" nodes with no tasks, so sometimes you may need to put a dummy task linked with the full run-time library onto a node that would otherwise have no tasks placed on it. This message will also be generated when one task has a virtual channel that goes to a processor where no task calls any of the channel I/O functions.

ERROR: URGENT or PRIORITY can only be used once per task

In each TASK statement, you may have one URGENT or one PRIORITY attribute only.

FATAL INTERNAL ERROR: message

An error has occurred in the internal working of the configurer. Please make a note of the message, which describes what has gone wrong, and get in touch with your dealer or 3L.

WARNING: adjusting priority of task task to max (largest permitted value for processor type type)

You have exceeded the maximum allowed value for the PRIORITY attribute for this type of processor. The configurer has revised the value down to the maximum.

WARNING: connection name cannot be made virtual

The configuration file explicitly requests that the named connection should be made virtual, but that cannot be done. For example, one or both of the tasks to be connected may be on a processor type for which Diamond presently does not support virtual channels.

WARNING: ignoring AVOID=area for processor name TYPE=type

The AVOID attribute of the PROCESSOR statement allows you to tell the configurer not to load anything into a particular physical memory area. However, the named processor is of a type that doesn't have that kind of memory area.

WARNING: link name[number] has been connected to itself

The two links mentioned in a WIRE statement are the same. This is probably an error.

WARNING: no memory allocation specified for task name: assuming rest of processor's memory

If none of the STACK, HEAP or DATA attributes is specified for a task, this message will be given, and the task will be treated as if you had written DATA=?

WARNING: PLACE and VIRTUAL conflict: connection name will be made PHYSICAL

This warning is issued if you attempt to place a virtual connection on a wire. The placement will be honoured, but the connection will be made PHYSICAL. This may lead to subsequent errors.

WARNING: PROCESSOR name BUFFERS outside acceptable range: using number

The processor called name has been given the BUFFERS=n attribute, but n is outside the allowed range of values, so the configurer has substituted a different number instead. The acceptable range of values is described in "BUFFERS Attribute".

WARNING: TASK name OPT=value invalid for type processors: ignored

The following are the logical area names allowed for the C6000: DATA, STACK, HEAP, CODE, STATIC, or the name of a section in the task image file.

WARNING: unconnected input port id bound to dummy channelWARNING: unconnected output port id bound to dummy channel

These two messages indicate that a task's port has neither been connected to another task nor bound to a value. This is not wrong, but may indicate an omission. The port has been bound to a dummy channel. Any operations using the port will never terminate.

WARNING: unknown special section name name in file filename ignored

The task that is being read from the specified file apparently requires some special resource that the configurer does not know about. This message should not normally happen.

WARNING: UPR BUFFERS outside acceptable range: using number

The value of n in a UPR BUFFERS=n statement is outside the allowed range of values.

WARNING: UPR MAX not multiple of four: using number

The value you gave in a UPR MAX=n statement was not a whole multiple of four (see "MAX attribute"). The configurer has substituted the value number instead.

WARNING: UPR MAX outside acceptable range: using number

The value you gave in a UPR MAX=n statement lies outside the range of acceptable values (see "MAX attribute"). The configurer has substituted the value number instead.

Chapter 9. The Server

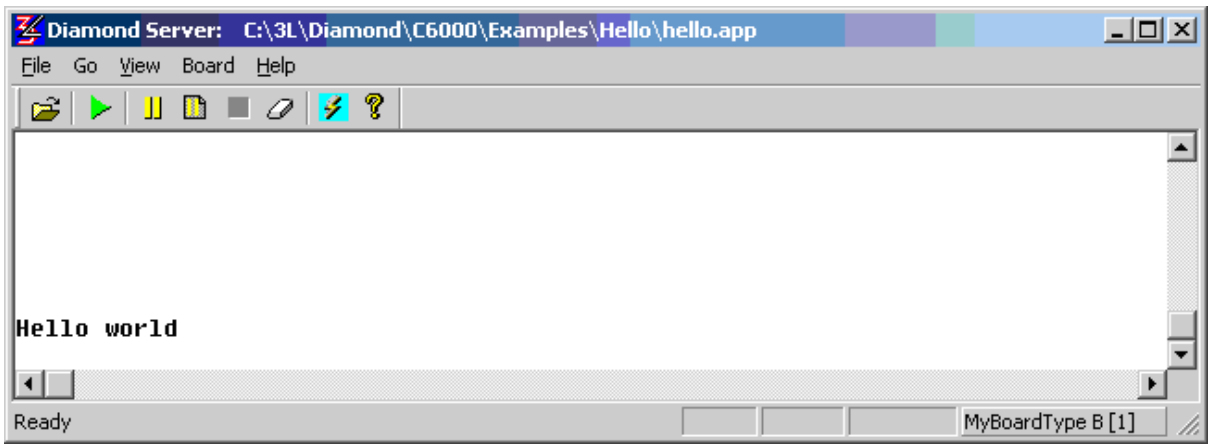
The server, WS3L.EXE, is the component of Diamond that provides the simplest way to load an application into a network of processors and allow it to communicate with the host.

Overview

The server is made up from three main components: the User Interface, the Server, and the Link Driver.

The User Interface

The User Interface provides a window that you can use to control the execution of your application. It provides controls for starting and stopping the application and an output area to display output sent to stdout or stderr.



The Server

`serve3L.dll` is the module that communicates with the DSP board and provides services for the running application. While this is usually accessed by the server's user interface, any program may use it to control Diamond applications. Ther are examples of this in **`<Diamond>\server\examples\`**.

The Board Interface

The Board Interface sits between the Server and the DSP board, and provides a read/write communication link between them. When the server starts running, it looks for all the board interfaces that have been installed on your system. If it finds only one or you have previously selected an interface, that one will be selected; otherwise, you will have to select the appropriate board interface yourself.

Starting the server

You can start the server in four ways:

1. By double-clicking on the file WS3L.EXE in the Diamond installation folder (or a shortcut to it). This will bring up the server window, but no application will be selected.

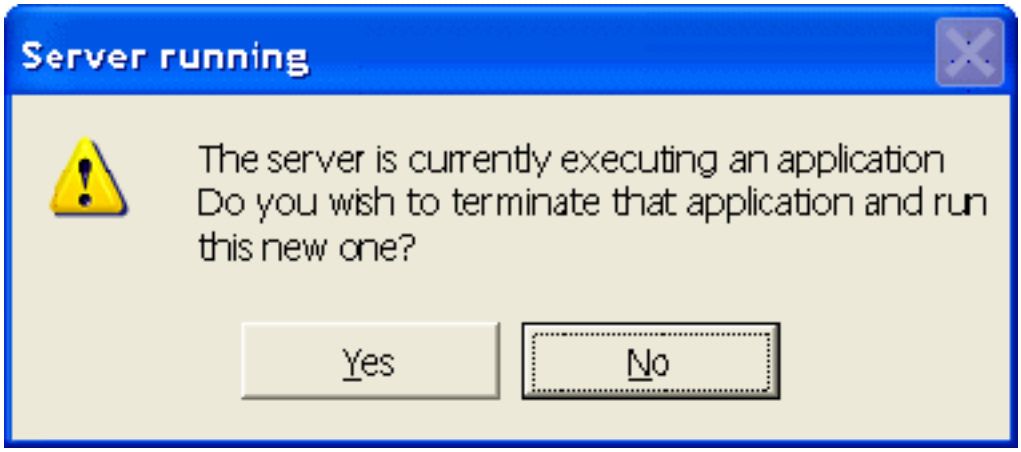
2. By double-clicking on a **`.app`** file. This will only work if you have associated WS3L with the **`.app`** file type. The Diamond installation procedure does this for you, but you can do it yourself at any time by following the operating system's instructions in "Start/Help/file types/associating extensions with". When the server starts running it will have selected the application file you double-clicked.

3. By dragging a **.app** file from an explorer window and dropping it into the server window.

4. By giving WS3L as a command at a DOS prompt. You can also give the name of an application file as an optional argument and the server will start running with that file selected. If you do not specify the .app file type, the server will append it automatically. Placing the optional argument "-exit" after the application file name will make the server terminate when the application terminates. This can be useful in batch files that invoke several applications in a sequence.

When the server is started with an application selected, it will attempt to load that application into your DSP network and start it running. This will only be possible if the server knows which DSP system you have; it knows this if it can only find one link interface when it is invoked. You can stop the server from running applications under these circumstances by unselecting **View/Options/Run .app files when double-clicked (or when started from DOS)** This is described further under Options.

The server will normally only allow a single instance of itself to be created; when you start the server, the last instance of it to be started will be used. If this instance is currently executing a program you will get the following alert:



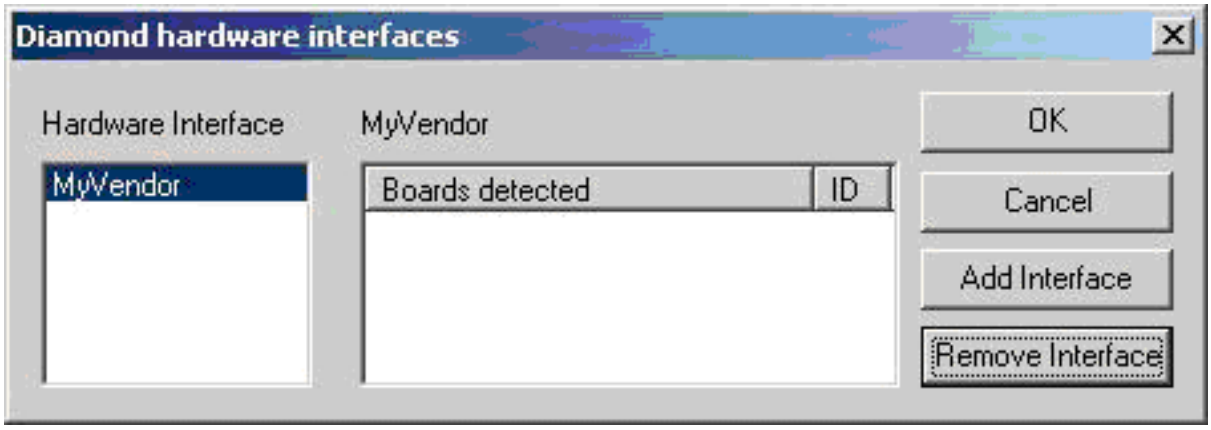
If you select "No", the default option, the existing application will continue to run and the request to run a new application will be ignored. If you select "Yes", the running application will be stopped and your new application started.

You can change the default behaviour by going to View/Options/Advanced and ticking the box for Allow multiple instances of the server. With this option selected a new instance will be created each time you start the server. Note that you will normally need to select a new board interface for each additional instance of the server.

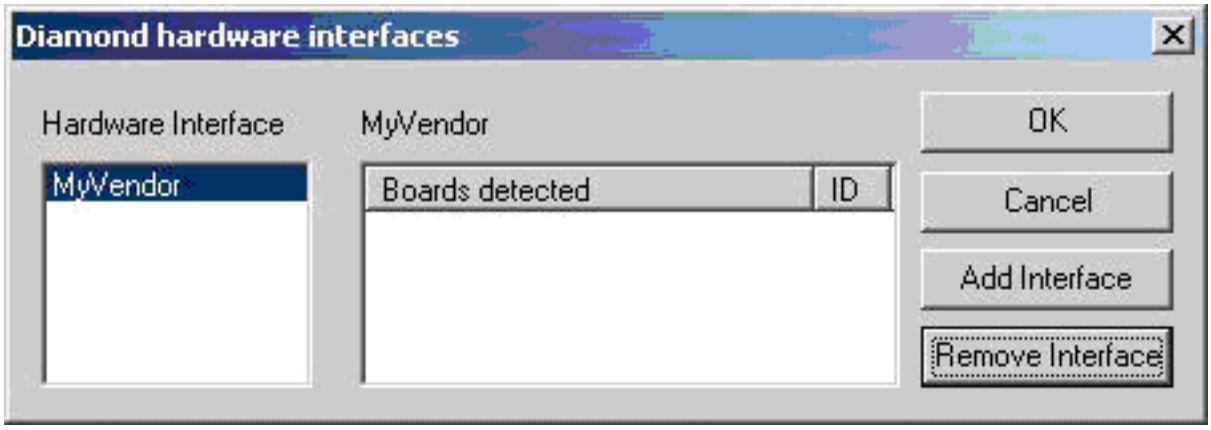
At any time you can create a new server instance selecting File/New Server Instance.

Selecting your DSP board

In order to communicate with your DSP system, the server needs to obtain a link interface. It finds link interfaces from hardware interfaces that have been added to the server. Each hardware interface can usually supply link interfaces for each of the DSP boards you have attached to your PC. Before you can run an application you must select the particular type of DSP hardware you want to use and from that select an interface for the DSP board that holds the root processor for your system. You do this by clicking on Board and then Select. This brings up a list of the installed hardware interfaces.



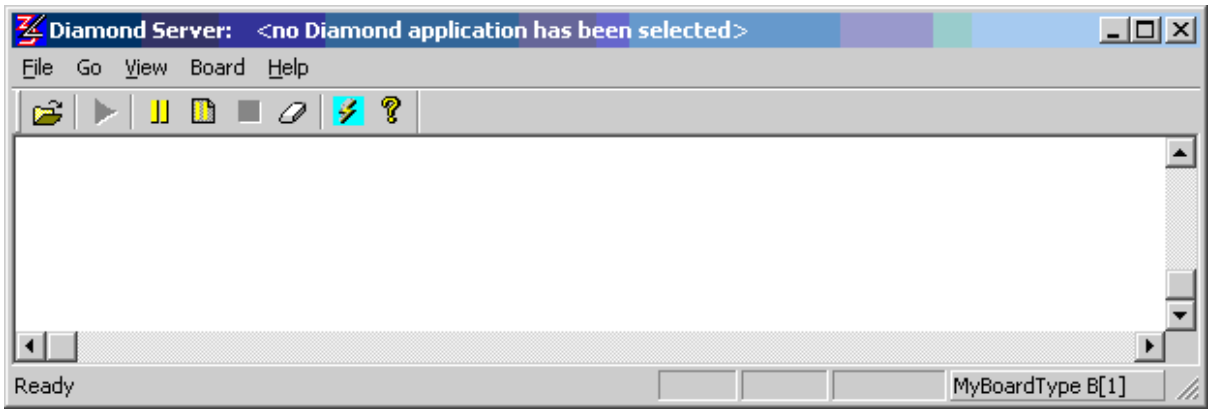
In the example above, only one hardware interface (MyVendor) has been added. Click on the hardware interface for the DSP board you wish to use and the interface will search for boards installed in your PC. A list of the detected boards will be displayed.



The ID is a hardware-dependent value returned by the hardware interface and can be used to identify the physical board found. The ID is commonly either an indication of the relative slot occupied by the board or the value of identification switches on the board. Consult your hardware manufacturer for further information.

Double-click on the appropriate board to select it, or click on the interface and then click OK. The name of the selected board will appear in an indicator at the bottom right of the server's main window with its ID value in square brackets.

The server will remember which interface and board you select, and will attempt to make the same selection for you again the next time it is started. Should the hardware have changed, the server will issue a warning and select the first board from the first hardware interface it finds.

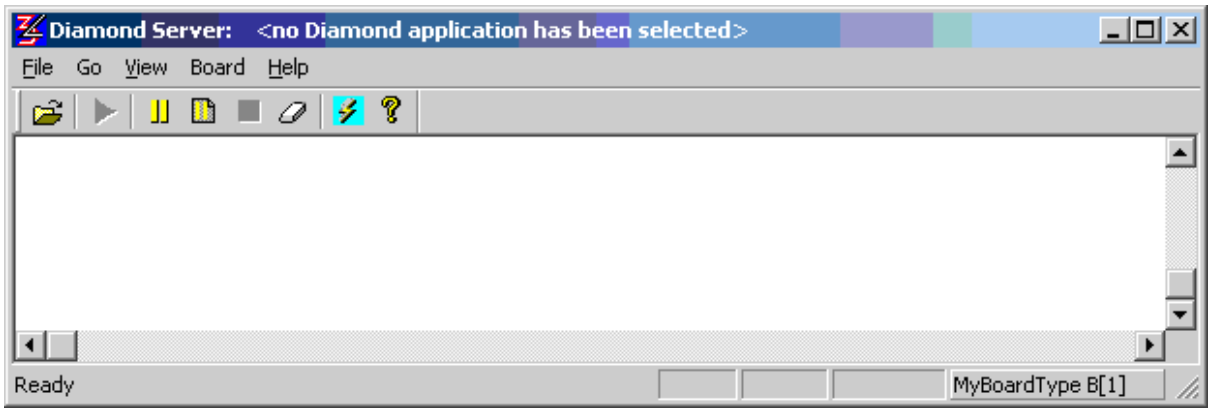


If you wish to install a custom interface you may add one to the list by clicking on New Interface. This will bring up a file selection window where you can navigate to the DLL defining the interface. Refer to Writing a board interface for details on creating new interfaces. You can also delete link interfaces from the list by selecting and clicking Delete. Several commonly-used interfaces are installed automatically each time the server is started.

Selecting an application

If you have started the server by double clicking on a Diamond application file, that file will be selected when the server starts. If you have started the server by double-clicking WS32.exe, or you want to change the selected application, click on File. This will give you a drop-down menu that lists the most recent application files that you have selected. Double-click one of these applications to select it. If the application you want to run is not listed there, click on Application and this will bring up a file selection window. You can also click on  to do the same thing.

When you have selected an application, its name will appear in the server's title bar.



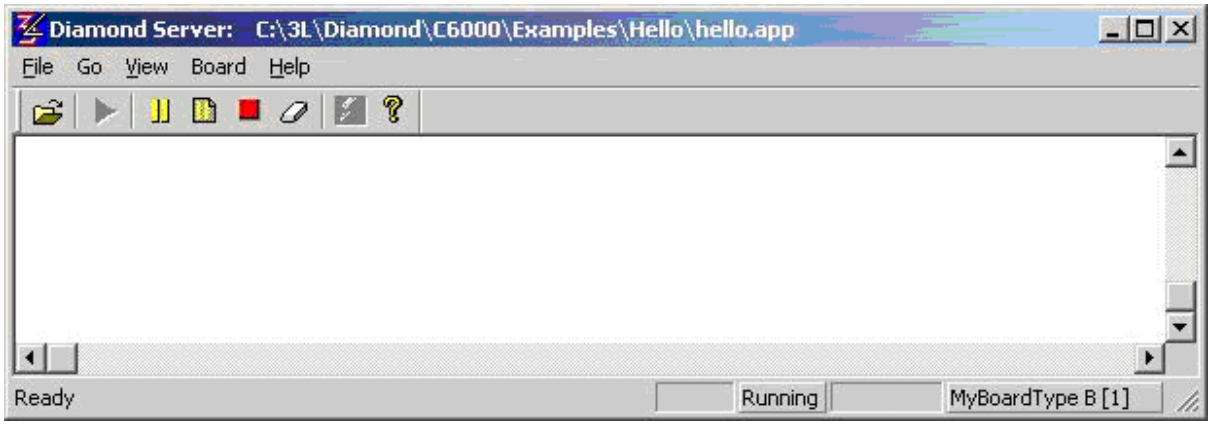
If the option Run application when selected has been ticked, the application will start running immediately.

Explicitly resetting the DSPs

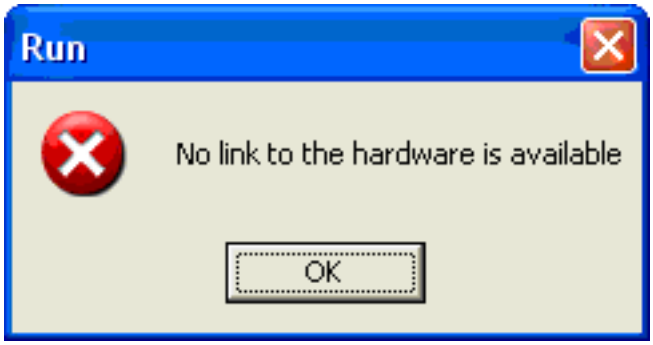
The server will usually reset the DSPs in your network before attempting to load an application. You can use  or Board/Reset to issue a reset to the network without loading an application.

Running the application

Once you have selected your DSP board, clicking on  or Go/Run will start your selected application. The start button will change to grey and an indicator at the bottom right of the main display will change to Running.



If the server cannot locate a link interface to use to talk to your board, you will get the following message:

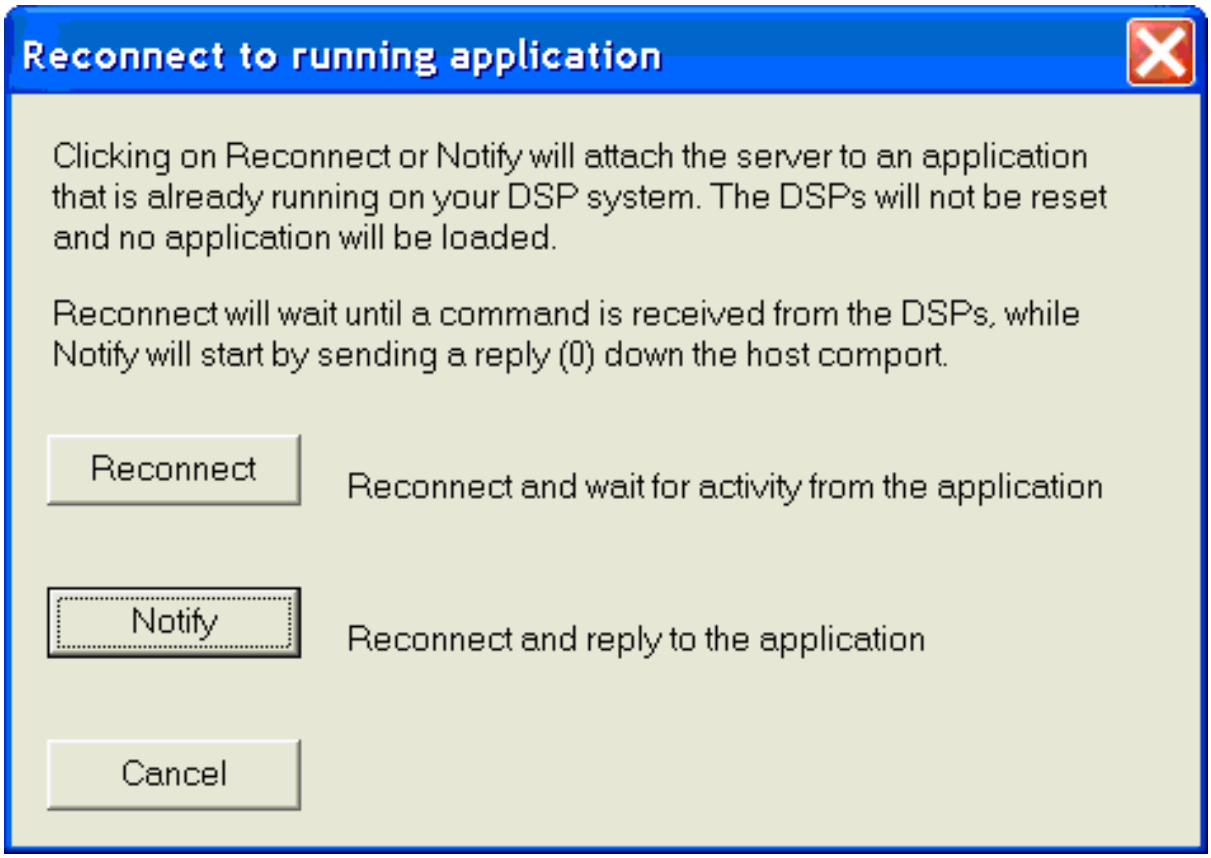


There are several possible causes for this:

- You do not have a suitable DSP board plugged into your PC;
- You have more than one board, but have not selected the particular one you want to use;
- You have another instance of the server running and that has already set up a link to your selected DSP board;
- Another application, possibly a maintenance utility from the board manufacturer, has claimed the DSP board.

Reconnecting the server

It is possible for an application to disconnect from the server and yet continue to run (see `disconnect_server`). The server can reconnect to such a running application at a later time using the Go/Reconnect menu option. This will bring up the following dialog:



Reconnect

connect and wait for the application to send a command.

Notify

connect and send a zero value to the server. This is usually seen as the response to a disconnect_server(1) call.

Cancel

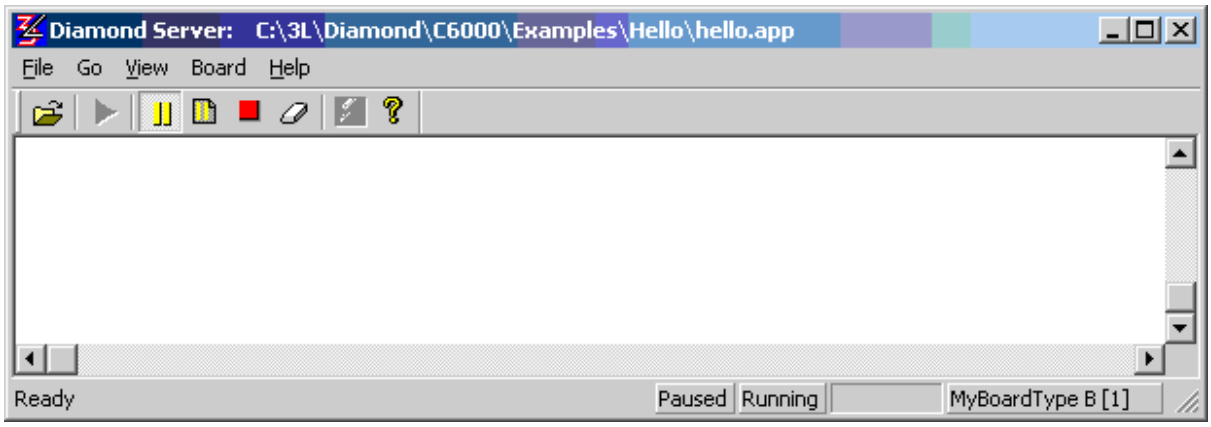
Ignore the reconnect request.

Stopping the application

You can stop the server from responding to your application by clicking  Note that this does not actually stop the DSP processors from running; it simply stops the server from listening to them.

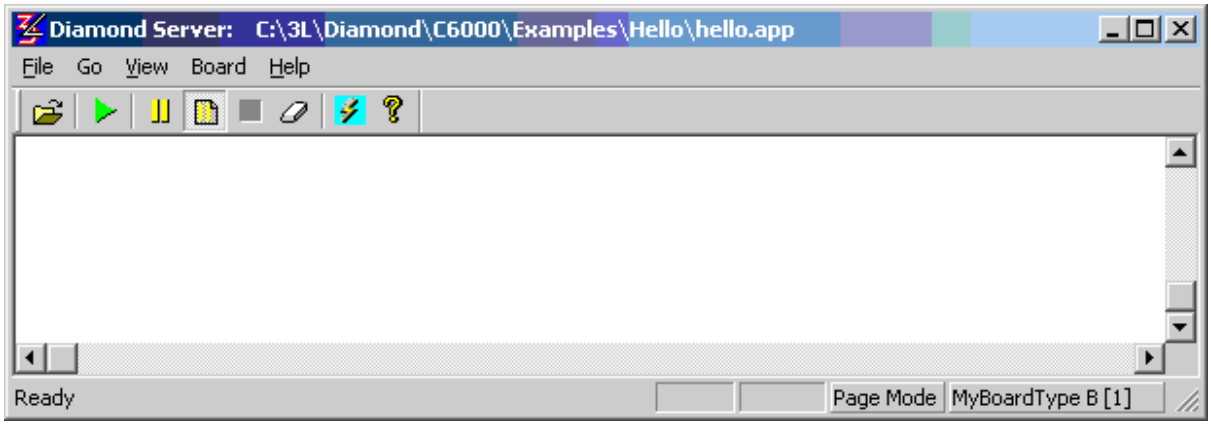
Pausing output

Clicking on  will pause the running application when it next tries to send more text to the output area. It will remain paused until you press  again to allow output to continue. An indicator at the bottom right of the window will show when output has been paused.



Page mode

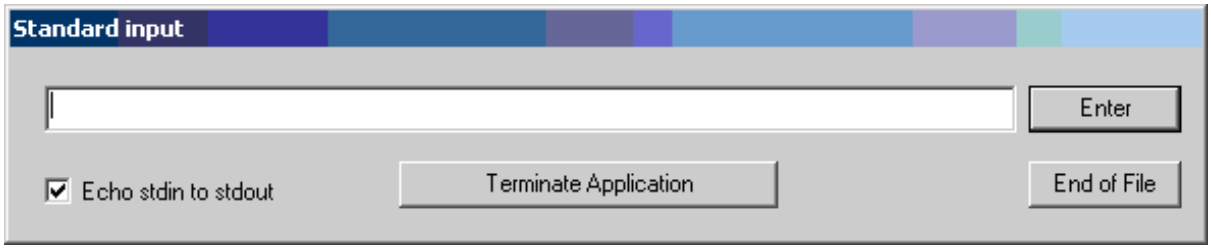
Clicking on  will select page mode and set the Page Mode indicator at the bottom right of the main window.



This will have the effect of pausing output every time the number of lines needed to fill the visible output area has been displayed. Clicking on will restart output for the next screenful. You can leave page mode by pressing again. Page mode is useful when you have an application that generates large amounts of output to stdout or stderr.

Input

When your application requests input from stdin, a "Standard input" window will appear:



You can type your line of text into the window and then hit either the Enter button or the Enter key on your keyboard. By default, everything you type will be echoed to the server's output area. Clearing the tick on Echo stdin to stdout will switch this off.

You can signal end of file by typing Ctrl+Z or clicking End of File.

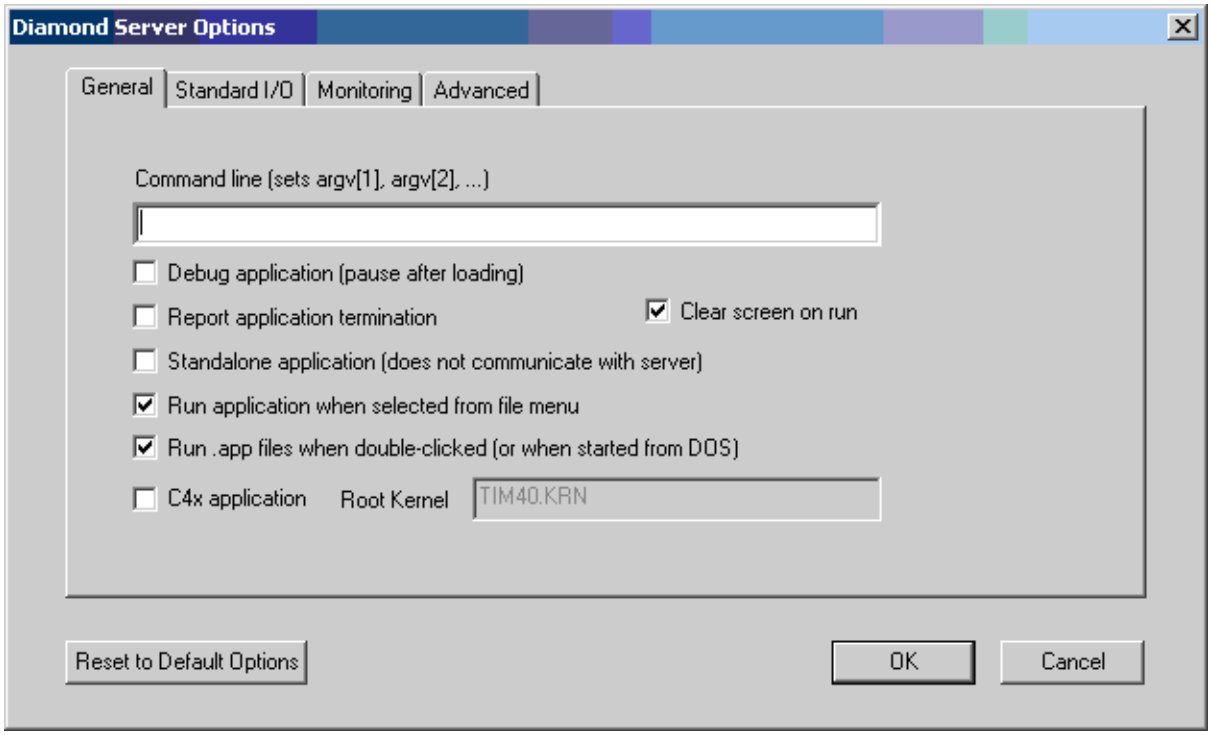
The Terminate Application button will both signal end of file to the application and stop the server from talking to it (see).

The window heading "Standard input" can be changed from the Diamond application with the `prompt` function.

Options

Clicking on View/Options will bring up the Server Options window.

View/Options/General Tab



Command line

The text you put in the Command line box will be broken into "words" and passed to your application, which can access them using argc and argv in the usual way. argv[0] will always be set to the full path for the application you have selected.

Debug application

Tick this box if you want to use Code Composer to debug your application. With this option selected, the server will pause after loading the application to allow you to pass control to Code Composer.

Report application termination

Tick this box if you want the server to bring up a notification when your application stops. This box will be ticked when the server is first installed on a PC.

Clear screen on run

Tick this box if you want the server to clear the output display each time you run an application.

Standalone application

Sometimes you may want the server to load an application but not attempt to communicate with it afterwards. Ticking this box will stop the server from attempting to communicate with an application once it has been loaded. Selecting this option makes the previous three options meaningless and will disable them.

Run application when selected

Tick this box if you want an application to be run as soon as you have selected it using File or . Clearing this option will mean you have to start the application explicitly using or Go/Run. This box will be ticked when the server is first installed on a PC.

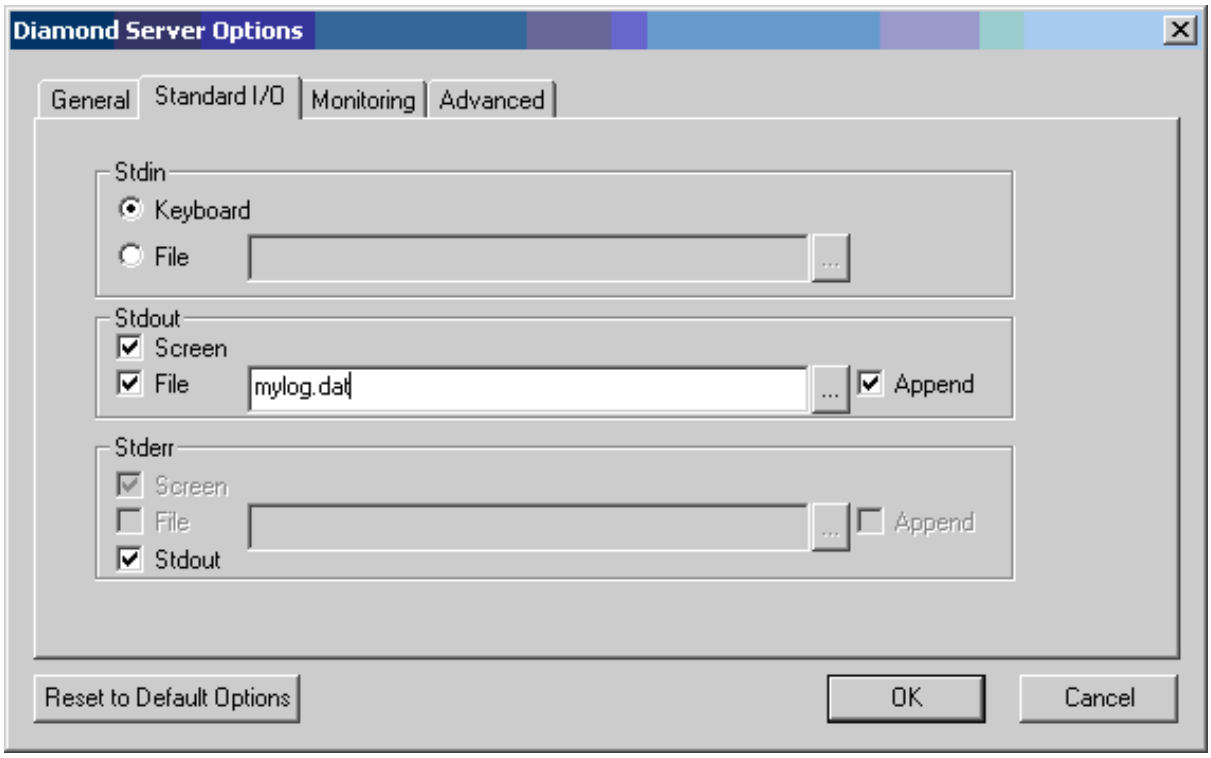
Run .app files

When you launch the server by double-clicking on a .app file or by giving a DOS command, the server will normally attempt to load and run the given application, providing that there is only one hardware interface installed and that can find only one DSP board. If you clear this box, the server will simple select the application but not run it.

C4x application

Tick this box if your root processor is a member of the C4x family. You should make sure that the "Root Kernel" box contains the full path of the root kernel file for your C4x board (usually TIM40.KRN). The value you place in here should be the same value as described under TISROOT in the Parallel C for the C4x User's Guide.

View/Options/Standard I/O Tab



This tab allows you to control the standard C streams, `stdin`, `stdout`, and `stderr`. Note that these settings only take effect at the point you start an application running; changing them during an application's execution will have no effect.

To specify a file, tick the appropriate File box and type the filename or press the button to the right of the file input box to bring up a browser.

stdin

You can make `stdin` take input from either the keyboard or a file. In the example above, input will come from the keyboard.

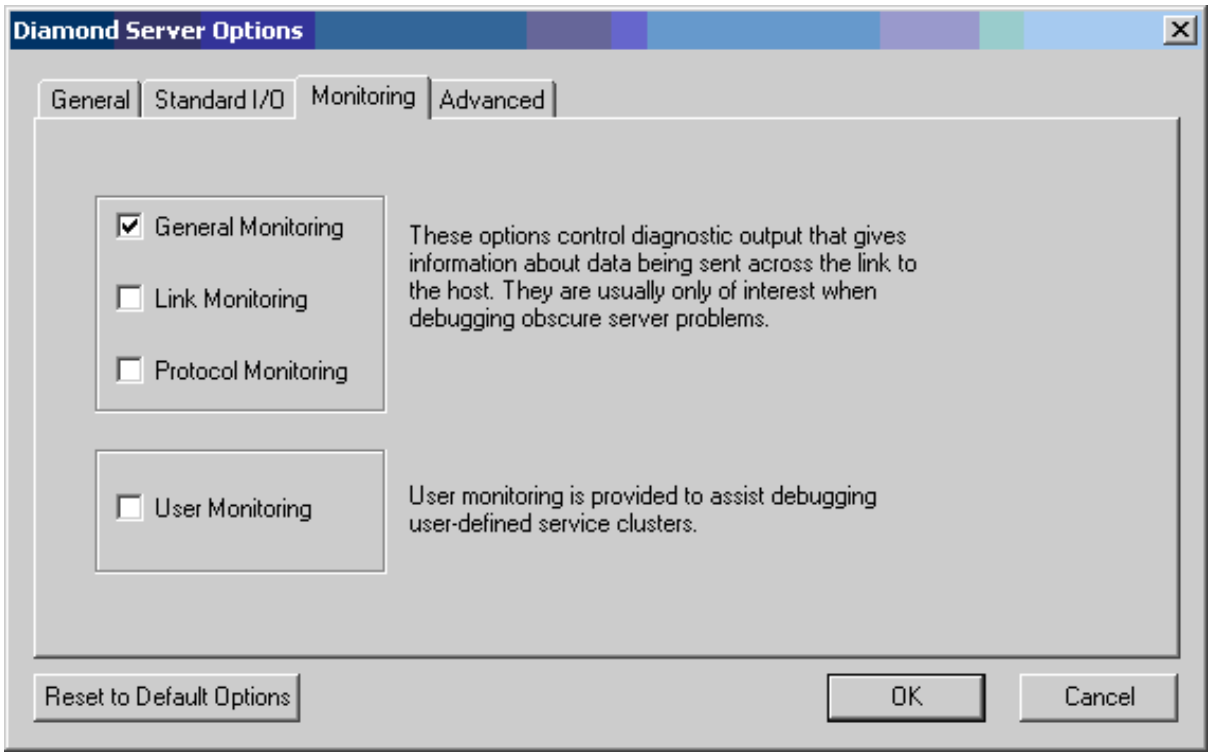
stdout

You can make output sent to `stdout` appear on the PC's screen, be sent to a file, or both. Ticking the Append box will cause any output to be sent to the end of the selected file; existing data in the file will not be changed. In the example above, everything sent to stdout will appear on the screen and will be appended to the end of the file "mylog.dat". Clearing both the Screen and File buttons would throw away all output sent to `stdout`. As this is unlikely to be what was wanted and could lead to great confusion, the server will silently reselect Screen on exit.

stderr

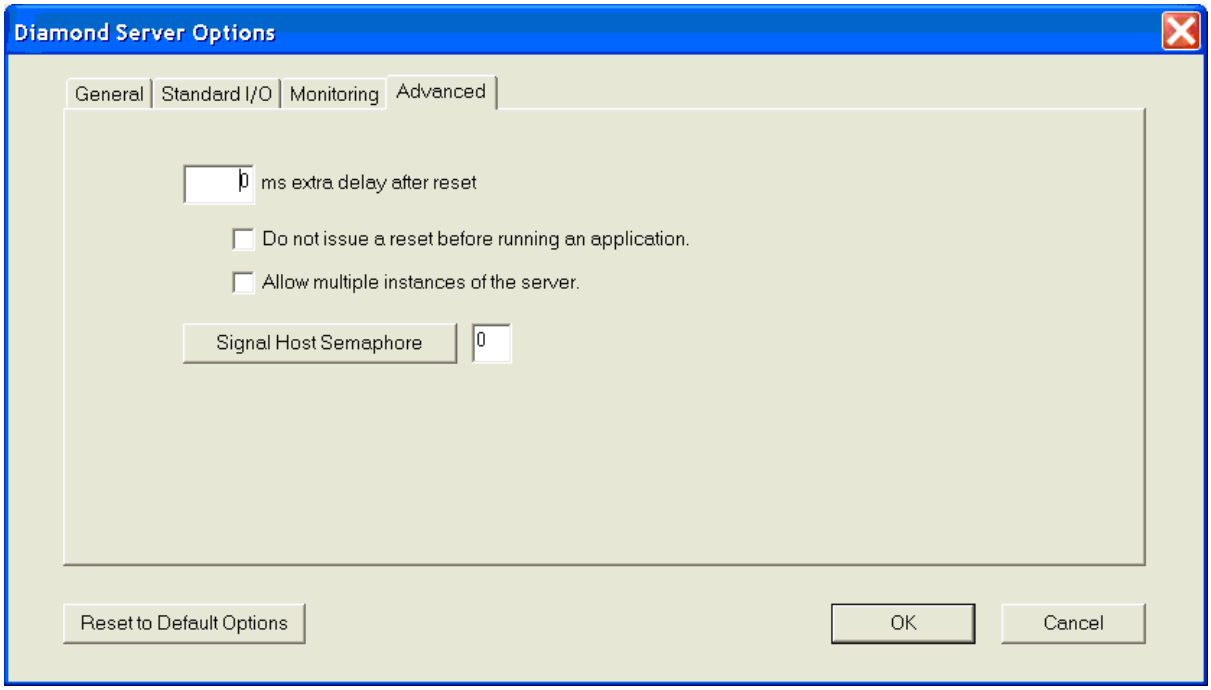
You can make output sent to `stderr` appear on the PC's screen, be sent to a file, or both. Ticking the Append box will cause any output to be sent to the end of the selected file; existing data in the file will not be changed. Alternatively, you can tick the Stdout box and have all references to `stderr` treated as if they were references to `stdout`. In the example above, everything sent to `stderr` will appear on the screen and will be appended to the end of the file "mylog.dat". Clearing both the Screen and File buttons would throw away all output sent to `stderr`. As this is unlikely to be what was wanted and could lead to great confusion, the server will silently reselect Screen on exit.

View/Options/Monitoring Tab



View/Options/Advanced Tab

The advanced tab provides unusual options that many users should never need.



ms extra delay after reset

In some DSP systems, the time the network takes to respond fully to a hardware reset depends factors that the server may not be able to determine, including:

- The number of processors present;
- I/O devices attached to processors;
- Manufacturer-defined processor options.

Normally the vendor's hardware interface will give the appropriate delay. This option can be used to make the server wait an extra number of milliseconds after a reset has been completed before it attempts to load your application.

Do not issue a reset before running an application.

Selecting this option will stop the server from resetting your DSP system before attempting to load an application. Pressing  will still reset the system. Note that most systems must be reset before they will load applications.

Allow multiple instances of the server.

When this box is ticked, every attempt to start the server will result in a new instance of WS3L.exe being created. The default state is for this box to be left unticked, so any running instance of the server will be used. Even with this box clear, you can start a new instance of the server by selecting File/New Server Instance.

Signal Host Semaphore

Send an asynchronous message to the root DSP to signal one of its Host semaphores. The particular semaphore is specified in the box to the right of the command button by an integer, n, in the range 0 <= n <= 10. Root tasks can wait on these semaphore using the library function host_sema_wait. Internal details of this mechanism are available here.

Reset to Default Options

This button can be used to set all the available options back to the default settings selected when the server is first installed.

Board properties

Some DSP boards have settings that you can adjust from the server. Selecting Board/Properties brings up a window that allows you to manipulate these settings. This option does nothing for boards that do not have changeable settings.

Help information

You can display this User Guide while running the server by clicking on Help/Diamond Help. If you have more than one edition of Diamond installed, you will be asked to select the appropriate User Guide.

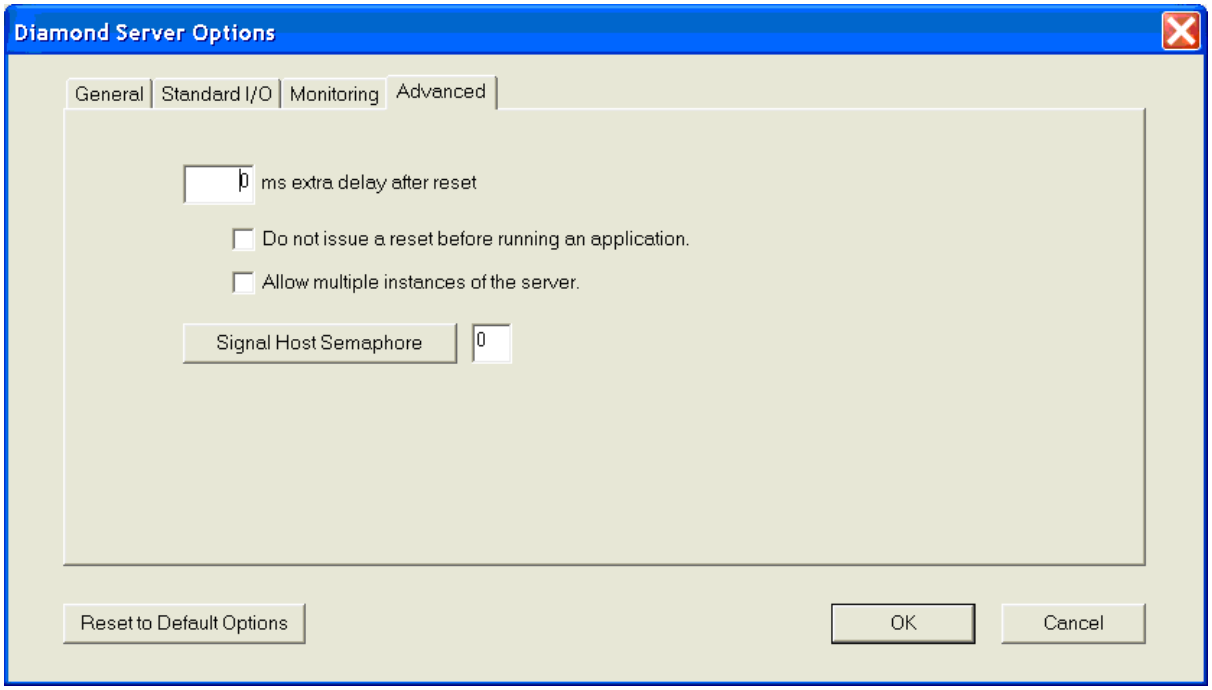
Shortcut keys

The following keys may be used in the server instead of using the mouse:

Key	**Button**		**Meaning**
F1		(Help/Diamond Help)	Display User Guide
F3		(File/Application)	Select application
F5		(Go/Run)	Run selected application
Shift+F5		(Go/Stop)	Stop application
F7		(View/Options)	Manage option settings
Alt+C		(View/Clear Screen)	Clear output display
Pause		(Go/Pause)	Toggle pause
Scroll Lock		(Go/Page Mode)	Toggle Page Mode

Server version

You can discover which version of Diamond you are running by clicking or selecting Help/About Server...



Error messages

Two classes of error are recognised by the server:

Software exceptions	The server reports these on behalf of system tasks located in the C6000 network. They are displayed in the following format:
	
	`*** Software exception: xxxxxxxx yyyyyyyy zzzz`
	`Processor=proc Severity=severity`
	`message`
	
	The server does not stop after receiving a software exception message from the network.
	
	severity — may be information, warning, error, fatal or unknown.
	
	proc — The processor number, proc, can be related back to a named processor in the application's configuration file using the configurer's "-m" (map) option switch.
	
	xxxxxxxx — is a 32-bit hexadecimal message code. At present, the only codes that describe user errors are 1102_{16} and 1202_{16}. All other codes are the result of internal software errors in system tasks, or of hardware link errors that have resulted in corrupt messages being sent to system tasks.

	yyyyyyyy and zzzzzzzz	are two further 32-bit hexadecimal values that give extra information about some message types.
	message	is only given for some message codes. When present, it is a textual version of the message code.
Server errors	These are detected by the server itself during its operation. When the server detects an error, it outputs an appropriate message and stops.	

Internal Details

This section gives details of the internal operation of the server. It is intended for users who want to add to the functions provided by the server or develop their own user interface.

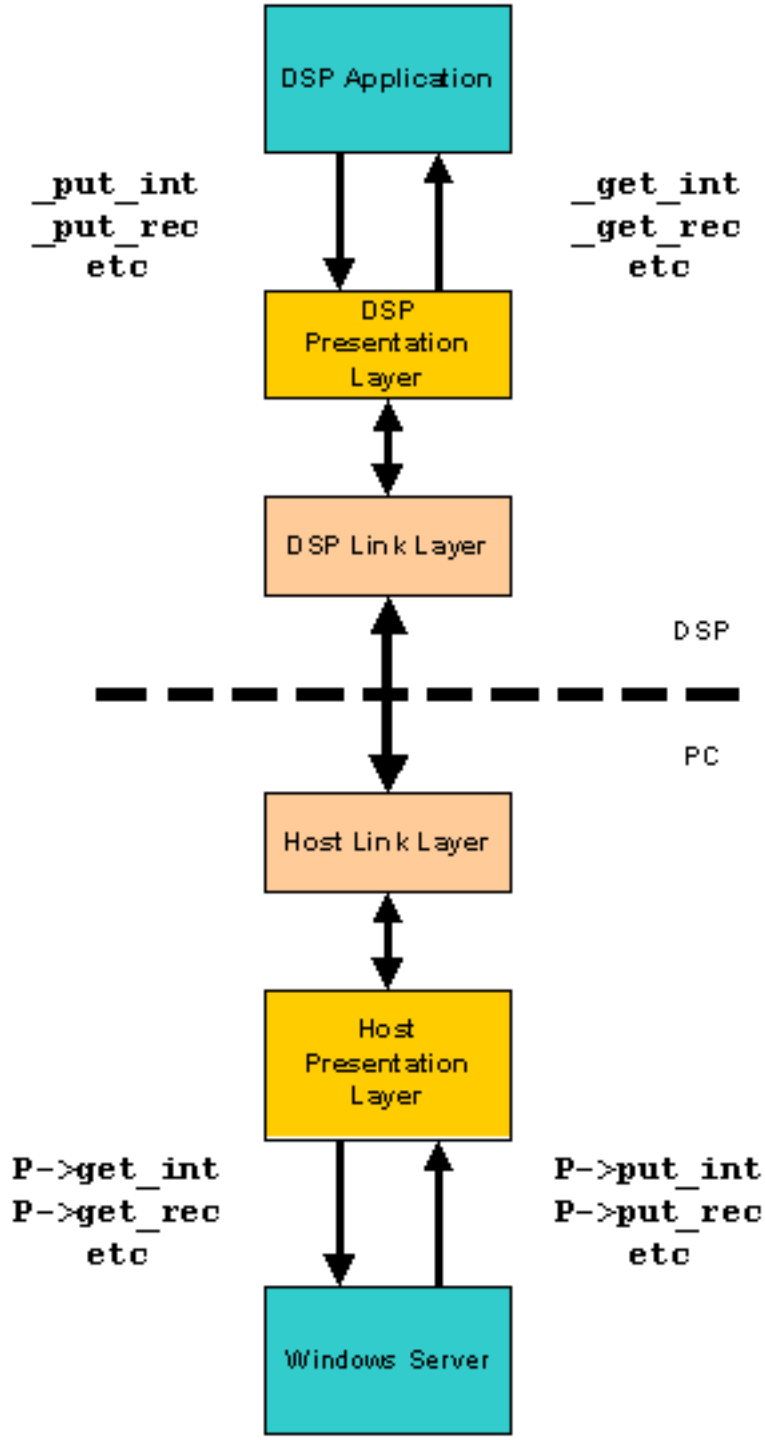
Loading applications

Apart from providing host services to a running application, the main task of the server is to load it into the processor network in the first place.

The loading process proceeds as follows.

1. The target system is reset. The link interface driver decides precisely what this means.
2. Code from the application file is then sent down to each processor. On some systems, this will be done indirectly, by sending all of the data down the host link to the root processor. This will deal with pieces relevant to the root and pass the remaining data down appropriate links to the other processors. On other systems, all of the processors will be loaded directly by the server.
3. Once the various loaders have loaded the code, they start all the tasks on their processors. Finally, the network sends to the server a ready-to-go message; this message can be inhibited by configuring with -A.

Server structure



The server is made up from a number of components, mainly three types of driver: the cluster drivers, the presentation layer drivers and the link interface drivers. This reflects the fact that the server works with a layered communication model; drivers at each of the levels use the functions supplied by the next level down (if one exists).

Cluster drivers

The cluster drivers correspond to the application layer of the ISO Reference Model. They supply services that are used by the Diamond libraries or by user-written code to perform standard I/O and other host functions. Each of the various clusters has a number, and when a program wishes to invoke a service, it transmits to the server a code that includes both the cluster number and the number of the service within that cluster. Following

the code can be parameter values for the service. When the service has been completed, the server will respond by sending back result parameters.

Presentation-layer drivers

This layer corresponds to the presentation layer of the ISO Reference Model. The presentation layer drivers provide functions like receive an integer or send a record, which enable the cluster drivers to communicate with the run-time libraries of the tasks in an application using common data representations. The presentation layer converts between host and target system formats, e.g., if there are different floating-point formats, or a different ordering of the octets within an integer.

Presentation protocol 5 (P5) is used for the C6000, and in some other implementations of Diamond. P1 is an older protocol used by previous 3L software.

The P5 protocol operates on the host by reading a parameter block from the DSP. This contains all the information needed by the service being called. The block is held in a buffer and accessed by the presentation input functions (`Present::get_int`, `Present::get_rec`, etc). When all of the information in the buffer has been taken, the same buffer is reused to collect information being sent back to the DSP by the output functions (`Present::put_int`, `Present::put_rec`, etc). The service must read everything it needs from the buffer before it attempts to return any values to the DSP. Usually, when the service returns, the server will finish by calling the presentation layer's push function to transmit the contents of the buffer back to the DSP. The server will then wait for the next service request.

The Presentation Interface (P)

Every cluster is given a pointer to a presentation layer, P. This provides the following member functions:

Present::get_int

```
ps1_integer Present::get_int();
```

Read a single 32-bit integer from the input buffer.

Present::put_int

```
void Present::put_int(ps1_integer);
```

Put a 32-bit integer to the output data buffer.

Present::get_rec

```
size_t Present::get_rec(void *);
```

Get a record from the input data buffer. A record is transmitted as an array of octet values. The values are placed in the object pointed at by the parameter, and the function returns the number of octets received.

Present::put_rec

```
void Present::put_rec(size_t, const void *);
```

Send a record to the output data buffer. A record is transmitted as an array of octet values. The first parameter gives the number of values and the second parameter locates the object containing them.

Present::get_double

```
ps1_double Present::get_double(void);
```

Get a floating-point double value from the input data buffer.

Present::put_double

```
void Present::put_double(ps1_double d);
```

Send a floating-point double value to the output data buffer.

Present::get_bits

```
ps1_bits Present::get_bits(void);
```

Get an uninterpreted four-octet value from the input data buffer. This is the same as get_int when the host and DSP intege formats are equivalent

Present::put_bits

```
void Present::put_bits(ps1_bits b);
```

Send an uninterpreted four-octet value to the output data buffer. This is the same as put_int when the host and DSP integer formats are equivalent.

Present::signal_host_sema

```
void Present::signal_host_sema(int n);
```

Signal an asynchronous event to the root processor. The root maintains 10 host semaphores, so the parameter must be in the range 0 <= n <= 9. The protocol for communicating between the host and the root puts the root in charge; the host is usually passive and waits to be told what to do. The root sends a service request and then waits for a reply. This function allows the host to break into this sequence and interrupt the root, making it signal one of the host semaphores. Threads in the root can wait for these semaphores using `host_sema_wait` and deal with the asynchronous event, rather than having to ask the host repeatedly if the event has occurred. For example, you may wish to be able to ask a running application to display some status information. This can be done by having a thread on the root that waits on a host semaphore. When the semaphore is signalled, the thread can print out the required information and then wait for the semaphore to be signalled again.

Note

Note that this function only signals a semaphore and does not transfer any data; all data transfers must be initiated by the root.

Present::signal_host_mask

```
void Present::signal_host_mask(unsigned int m);
```

Call signal_host_sema for each bit set in a bitmask. The function takes a bitmask, m, and calls `Present::signal_host_sema`(n) for each bit (1<<n) set in the mask. So, signal_host_mask(5) will call `Present::signal_host_sema`(0) and `Present::signal_host_sema`(2). As

there are only 10 host events, all bits in the mask other than the least-significant 10 must be 0

Present::push

```
void Present::push(void);
```

Transmit the data buffer to the root DSP. This function must always be called before the server returns to read in the next command word.

Link-interface drivers

This layer corresponds to the ISO Reference Model's data link layer. In general, different C6000 boards have different host interface hardware. For each kind, there is a link interface driver, which provides a common set of low-level functions to the presentation layer drivers. These drivers are usually provided by your hardware manufacturer and are listed in the Board Selection menu.

Extending the server

As well as providing access to graphical user interface (GUI) features, the 3L Windows Server allows you to define your own host-based services and make them available to the C6000. You do this by defining your own cluster of services and building them into a dynamic-link library. When the server sees a request for the cluster, it attempts to load the relevant library and use the services it provides.

Locating clusters

The clusters required by the Diamond run-time library are built into the server. These clusters are installed when the server starts by being inserted into a table that is efficiently searched each time a service is required. If a cluster cannot be located in this table, the server attempts to find it externally.

You can use the server object's member function SetCallback to install a single call-back handler for the server to invoke when a call is made to any unknown cluster. The base class for a call-back is as follows:

```
class Callback {
   public:
      virtual bool Handler(unsigned int  ClusterNo,
                           unsigned int  ServiceNo,
                           bool          ReplyWanted,
                           Present      *P)=0;
};
```

Once you have installed a call-back, its handler member will be called when an unknown cluster is referenced. It can either handle the service request or install an appropriate cluster handler. If the handler returns true, the server will flush any pending output requests to the root by calling P->push() and then continue to process the next command; subsequent references to the cluster will result in direct calls to the handler. If the handler returns false, the server will assume that a new service cluster has been installed and will attempt to locate it; once located, the cluster will be entered directly on subsequent references without invoking the call-back.

If the request has still not been satisfied, the server attempts to install a dynamic-link library for the cluster. This library is found by searching for the file `Clu3L_xx.dll`, where xx is the hexadecimal representation of the required cluster number. This will be a minimum of two characters long. For example, given a command word of 12345678_{16}, the cluster number is 3456_{16} and the server will try to load `Clu3L_3456.dll`.

The server will return a value of 00000080_{16} if both of these methods fail. In addition, if you have selected General Monitoring, the server will display a warning message. When using your own cluster, the DSP should start by checking that the services can be located by attempting to call service 0 in the cluster. This will return 0 if the cluster has been located:

```
int CheckCluster(unsigned int clu)
{
   int r;
   _ps1_put_bits(clu<<8);    // service 0 exists in all clusters
   r = _ps1_get_integer();
   if (r != 0) printf("Missing cluster %d\n", clu);
   return r;
}
```

Each cluster is given pointers to two classes:

P This is a pointer to the Presentation class, used for all communication between the server and the root DSP.

C This is a pointer to the Core class which provides commonly-used functions.

Server Operation

The server module operates as a thread, started by the Server User Interface. It runs in a loop that reads a single 32-bit service word from the selected host link and uses this to dispatch to the appropriate cluster.

The service word has the following form:

31	**30–24**	**23–8**	**7–0**
NoReply		Cluster Number	Command Number

On return from cluster, the server optionally sends a reply word back to the root DSP and waits to read the next service word. The NoReply bit is set if the DSP does not expect any reply from the command. All of th communication is done through functions defined in the Presentation Interface. The server maintains a pointer to this class (P).

```
Running = true;
while (Running) {
   unsigned int Service    = P->get_bits();
   unsigned int CommandNo  = Service&255;
   unsigned int ClusterNo  = (Service>>8)&0xFFFF;
   C->NoReply              = Service>>31;

   Cluster *cl = {locate the cluster}

   unsigned int Answer = cl->Call(ClusterNo, CommandNo);
   if (C->NoReply==0) {
      P->put_int(Answer);   // issue reply
      P->push();            // send any pending output to ROOT
   }
}
```

The loop terminates when Running gets set false. This is done either by the standard stop service from the root or by the user interface on the host calling the server member function StopRunning. Note that this cannot take effect until the current iteration of the loop has completed; stopping the server when it is stuck or waiting for a response from the root has to be done using TerminateServer.

Note that services explicitly requesting NoReply cannot pass back any parameters. It is possible for service functions to set NoReply, leaving it the responsibility of those functions to send any necessary replies and end by calling P->push().

Building your own cluster

The Diamond installation provides an example of a user-defined cluster in the folder `<Diamond>\server\cluster`. This builds a library, **`Clu3L_0C.dll`**, to support references to cluster 12 ($0C_{16}$). You can take this code as a base and develop your own cluster from it. The cluster is set up as a Microsoft Visual C++ project, with workspace ExCluster.dsw.

1. Start by selecting a name for your new service cluster. You should take copies of the relevant files from the example and rename them accordingly.

2. Select a Cluster Number for your new service cluster: Any application may use cluster number 2 as an End-User cluster. This service will never be used by 3L or third-party clusters. Cluster 3 is similarly reserved as a Prototyping cluster. If you intend supplying your service cluster to other users of Diamond as a third-party cluster, you should develop it using cluster 3 and then apply to 3L for a Registered Service Cluster before shipping to customers. Cluster 12 is reserved for the example cluster.

3. Define a header file for your cluster based on ExCluster.h and change:

   a. CLASS. This name will be given to the class that will define your cluster. The example uses Ex_Cluster.

   b. CLUSTER_ID. This C-style string is returned in exceptions thrown by the cluster. The example uses "Example Cluster".

   c. The list of the services that your cluster will provide. This is actually a list of the names of the member functions in the class that you are going to create. Each entry is of the form:

      ```
      REF_SERVICE(<member function name>)
      ```

      For example: REF_SERVICE(Square)

4. Define the entry point for your library based on the file Example.cpp. The only change you need to make in your version is to replace "ExCluster.h" with the name of the header file created in step 2 above.

5. Define the services your cluster will provide based on the file ExCluster.cpp. You need to change 3 things:

   a. Replace "ExCluster.h" with your header defined in step 2

   b. Define the member functions that will implement your services. These will correspond with the references specified in step 2.c above. Each service is defined as in the following example:

      ```
      DEF_SERVICE(Square)
      {
         int value = P->get_int();
         return value*value;
      }
      ```

      This defines a member function Square that will take a single integer value from the parameter stream (get_int) and return the square of that value. The way services communicate with the server is discussed here. The example defines three member functions, but you may define any number, although defining no services would be very strange! The functions communicate with the root DSP using the presentation interface pointer, P. They may also use common functions accessed through the pointer C.

6. Define the mapping of command numbers to member functions by creating the array Services. When the DSP requests command "n" from this cluster, the member function identified by element "n" in this array will be executed. Each element must be one of:

   a. ENTRY(<service name>)

      <service name> must be the name of the member function to be called when the corresponding command is requested. For example, if you want the function Square, defined above, to be called on command 6, you would set element 6 of the array to ENTRY(Square).

b. NO_ENTRY

This is used as a dummy entry when a command number has no corresponding member function. Requesting this service will result in a no_service exception being thrown. Note that element 0 is always a null service and that the size of the Services array is derived automatically.

7. You should now build your cluster and create your library. The library name should be "Clu3Lxx.dll", where xx is the hexadecimal representation of the cluster number you have selected.

8. The server must be able to locate your library. You should put your new library in one of the following places:

   - The folder from which the server is loaded, usually `\3L\Diamond\bin.`
   - The current directory.
   - The Windows system folder, usually `C:\WINNT\System32`.
   - A folder listed in the PATH environment variable.

   Refer to the Windows documentation on LoadLibrary for further details.

Accessing your cluster from the DSP

The Diamond library includes functions to allow you to communicate with your cluster. These correspond directly to the functions in the server's presentation-level interface, P. Note that the DSP library automatically does the equivalent of the host's push operation when it detects you switching from writing (put…) to reading (get…).

Also note that you must ensure that no other threads attempt to access the presentation layer during a complete put…get sequence. This can be done by protecting the sequence with the `par_sema`.

```
#include <filer.h>

typedef int          ps1_integer;
typedef unsigned int ps1_bits;
typedef double       ps1_double;
```

_ps1_get_integer

```
ps1_integer _ps1_get_integer();
```

Read a single 32-bit integer from the input buffer.

_ps1_put_integer

```
void _ps1_put_integer(ps1_integer value);
```

Put a 32-bit integer to the output data buffer.

_ps1_get_bits

```
ps1_bits _ps1_get_bits();
```

Read a single 32-bit uninterpreted value from the input buffer. This is the same as _ps1_get_integer

when the host and DSP integer formats are equivalent.

_ps1_put_bits

```
void _ps1_put_bits(ps1_bits value);
```

Put a 32-bit uninterpreted value to the output data buffer. This is the same as _ps1_put_integer when the host and DSP integer formats are equivalent.

_ps1_get_double

```
ps1_double _ps1_get_double(void);
```

Get a floating-point double value from the input data buffer.

_ps1_put_double

```
void _ps1_put_double(ps1_double value);
```

Send a floating-point double value to the output data buffer.

_ps1_get_record

```
size_t _ps1_get_record(void *buf);
```

Get a record from the input data buffer. A record is transmitted as an array of octet values. The values are placed in the object pointed at by buf, and the function returns the number of octets received.

_ps1_put_record

```
void _ps1_put_record(size_t len, void *buf);
```

Send a record to the output data buffer. A record is transmitted as an array of octet values. The first parameter gives the number of values and the second parameter locates the object containing them.

_get_bits

```
ps1_bits _get_bits(void);
```

Get an uninterpreted 32-bit value from the input data buffer.

_put_bits

```
void _put_bits(ps1_bits b);
```

Send an uninterpreted 32-bit value to the output data buffer. This is often equivalent to _put_int when the host and DSP integer formats are equivalent.

The following is an example of how the DSP would call service 1 in the example cluster (12).

```
int reply;
#define COMMAND ((12<<8) | 1)
char text[] = "This is a message";
_put_bits(COMMAND);                  // the service word
_put_record(sizeof(text)+1, text);  // +1 for terminating 0
reply = _ps1_get_integer();          // send output and read reply
```

The Core Interface (C)

Core::Version

```
const char  *Core::Version(void);
```

Return a string showing the version of the server that is running. A typical value would be "Windows Server V2.6".

Core::Monitor

```
void  Core::Monitor(const char *format, ...);
```

Output a monitoring message to the server's output window. The message will always start with "MON: ". The call may be thought of as equivalent to:

```
fprintf(stderr, format,…);
```

Core::Quit

```
void Core::Quit(const char *format, ...);
```

Terminate the running Diamond application after printing a message to the server's output window. The message will start with "Server terminated: " and is generated as though the call were:

```
fprintf(stderr, format, …);
```

Core::Dump

```
void Core::Dump(const char *heading, void *b, unsigned int n);
```

Output a sequence of n 32-bit integer values, taken from b, in hexadecimal to stderr. The sequence will be introduced with the string "MON: heading n words".

Core::SetCommandLine

```
int Core::GetCommandLineMax(void);
```

Copy the given string into the command line parameter.

Core::GetCommandLine

```
const char  *Core::GetCommandLine();
```

Return a pointer to the command line parameter.

Core::GetCommandLineMax

```
int Core::GetCommandLineMax(void);
```

Return the maximum number of characters that can be put into the command line parameter.

The following five functions manipulate the arguments, argv and argc, that will be passed to the application when it is started. The arguments are considered to be in two parts: a verb and the rest. The verb will be made available as argv[0] while the rest will be broken down into "words" and made available through argv[1]… The total number of words, plus one for the verb, is passed as argc. The most common situation is when the verb is the application file being executed and the rest is the string set as the command line parameter.

Core::SetVerb

```
void Core::SetVerb(const char *v);
```

Define the string that will be used to initialise argv[0].

Core::SetRest

```
void Core::SetRest(const char *r);
```

Define the string that will be used to initialise argv[1..argc].

Core::GetVerb

```
const char *Core::GetVerb(void);
```

Return a pointer to the string to be used to set argv[0].

Core::GetRest

```
const char *Core::GetVerb(void);
```

Return a pointer to the string that will be used to initialise argv[1..argc].

Core::FreeArgs

```
void Core::FreeArgs(void);
```

Delete any strings set by SetVerb and SetRest; after this, argv and argc will be derived from the command line parameter.

Core::GetBootFile

```
const char *Core::GetBootFile(void);
```

Return a pointer to the currently-selected application file name.

Core::SetResultCode

```
void Core::SetResultCode(int n);
```

Set a 32-bit value that can be used to control program termination.

Core::GetResultCode

```
int Core::GetResultCode(int n);
```

Return the result code, set either by an explicit call on SetResultCode or by the user program calling exit(n).

Core::Output

```
void Core::Output(char *buffer, int length, int is_error);
```

Send length bytes from buffer to either stdout (is_error == 0) or stderr (is_error!=0).

Core::ReadLine

```
int Core::ReadLine(char *s, int max);
```

Read a line of characters from either the input prompt or a file, depending on the current stdin settings. The function will read up to max characters and return the number of characters read. The characters will be terminated with a zero char, not included in the number of characters. See fgets.

Core::OpenLogFiles

```
void Core::OpenLogFiles(void);
```

Activate the files associated with stdin, stdout, and stderr.

Core::CloseLogFiles

```
void Core::CloseLogFiles(void);
```

Deactivate and close all the files associated with `stdin`, `stdout`, and `stderr`.

Core::StopRunning

```
void Core::StopRunning(void);
```

Signal the server to stop responding to requests from the DSP. Note that the server will only stop when the next command from the DSP has been completed.

Core::IsRunning

```
bool Core::IsRunning(void);
```

Returns true if a program is running in the DSP and the server is able to process commands from it; returns false otherwise.

Core::G

```
SerCom *Core::G;
```

A pointer to the SerCom object the server is currently using to communicate with the GUI.

Core::Opt

```
Options3L *Core::Opt;
```

A pointer to the active Options3L object.

Core::NoReply

```
int Core::NoReply;
```

This flag is set when the NoReply bit is set in a command from the DSP.

Writing a board interface

Please contact 3L if you need to write your own board interface.

Replacing the Server GUI

The server runs as a separate thread to allow the user interface to operate independently, and all of its functionality can be accessed from host C++ programs. The header files for server access can be found in `<Diamond>\Server\Include`. and two sample programs can be found in:

`<Diamond>\Server\Examples\TIS\`	DOS application
`<Diamond>\Server\Examples\WinApp\`	Windows application

The interface to the server is in the library Serve3L.dll and this can be accessed as follows:

```
#include "Serve3L.h"

CServe3L *GetServer(void)
{
   HINSTANCE Dll = LoadLibrary("Serve3L.dll");
   if (!Dll) {
      MessageBox(0, "Cannot access Server3L.dll",
                    "Fatal Error", MB_OK);
      return NULL;
   }
   FARPROC Get = GetProcAddress(Dll, "Get3LServer");
```

```
   if (!Get) {
      MessageBox(0, "Get3LServer not in library Server3L.dll",
                    "Fatal Error", MB_OK);
      return NULL;
   }
   typedef void *((*GetType)(int));
   return (CServe3L *)((GetType)Get)(0);
}
```

The steps needed to run an application are as follows:

1. Get a pointer to the server object:

   ```
   CServe3L *S = GetServer();                // NULL on error
   ```

2. (optional) Bring up a dialog to allow the user to set server options:

   ```
   S->OM->RequestOptions();
   ```

3. Create an object to allow the server to communicate with this code:

   ```
   G = new Com(this);                        // see below
   S->Initialise(G);
   ```

4. Get a pointer to the board's Link object. This can be done in two ways.

   a. Ask the user to select a board:

      ```
      S->LM->LinkDialog(0);
      Link *L = S->LM->SelectedLink();       // NULL on error
      ```

   b. Look for a specific board:

      ```
      Link *L = S->LM->LocateLink("XYZ", 0); // NULL on error
      ```

5. Notify the server which link is to be used to access the DSPs:

   ```
   S->UseLink(L);
   ```

6. (optional) Set a call-back to deal with special clusters. See WinApp for an example of this.

7. Define the application to be loaded:

   ```
   S->SetAppFile("my.app");
   ```

8. Reset the processors:

   ```
   S->Reset(1);
   ```

9. Load the application and start the DSPs running:

   ```
   S->LoadAndGo();
   ```

10. When the application terminates, the communication object's Terminated member will be called, G->Terminated(), see step 3.

The Communication Object

The server communicates with the host using a communication object derived from the SerCom class (see `<Diamond>Server\Include\SerCom.h` for more information):

```
class Com : public SerCom {
      public:
         Com() {};    // add parameters if needed
         virtual      ~Com() {};
         virtual void  Output(char *buffer, bool error);
               virtual int   Input (char *buffer, int max);
         virtual void  Terminated();
         virtual void *Hook(int fn, void *p);
         virtual HWND  MainWindowHandle();
         virtual void  Init();
         virtual void  Exit();
         virtual void  Prompt(const char *string);
      private:
         // Add any application-specific private data here
   };
```

The host should create a single object of this class and make it available to the server (see step 3 above). The server will call the members of this object when necessary.

For example:

```
   void Com::Output(char *buffer, bool error)
   {
      // This is called when the DSP outputs to stdout or stderr.
      // error is true for stderr, false for stdout.
      MessageBox(0, Buffer, (error ? "Error" : "Output"), MB_OK);
   }
```

Replacing the Server

In some circumstances it may be necessary to dispense with the server interface altogether and control a Diamond application completely from user code on the host. In order to achieve this you need to do the following:

1. Build your application out of stand-alone tasks; as there is no host server, the application must not attempt to communicate with one. If you really do want to interface to the (complex) protocol used by the Diamond server, contact 3L for further details.

2. Specify the **`-A`** switch when you configure the application to prevent the loader from asking for a go-ahead message from the server. This is used to pause the application when debugging.

3. Each processor can communicate with the processor from which it was loaded using the library functions, `_host_in` and `_host_out`. The root can use these functions to talk to the host PC.

The host needs access to the link connected to the root processor; this is usually made available through a DLL supplied by the DSP board supplier. This DLL usually includes a function to load a Diamond application into the DSP network.

Chapter 10. The Diamond Library

Introduction

This chapter describes: Diamond's implementation of the ANSI C run-time library functions, as described in chapter 4 of the standard; some functions included for compatibility with other implementations of C; functions supplied by 3L to support Diamond's special multi-processing facilities.

Implementation-specific library functions are described in other chapters.

Format of Synopses

The function synopses indicate how to call library functions. Information about required argument types and function result types is presented in the form of a C function declaration prefixed by #include statements which indicate which headers, if any, must be used in order to access the function.

Flags

The following signs are used to flag the entries for certain functions.

DOS	Indicates that the function is specific to the MS-DOS operating system.
Heap	Indicates a function that manipulates the heap. References to these functions must be protected from references to other such functions in threads from the same task. See the discussion of `par_sema`. See also Server
Stand-alone	Indicates that the function is available as part of the stand-alone library as well as the full library. It may therefore be used by a stand-alone task. Functions without the stand-alone mark may only be used by tasks linked with the full library.
Server	Indicates a function that communicates with the server on the host. References to these functions must be protected from references to other such functions in threads from the same task. See the discussion of `par_sema`. See also Heap

Note

NUL is used here to indicate a character value of zero, such as used to terminate character strings. NULL, defined in `<stddef.h>` and several other headers, represents a generic "null pointer" value.

Headers

/Most functions in the Diamond library are accessed through header files in the Diamond installation folder.

Caution

It is important that you use these headers and do not attempt to use headers that may be provided by Texas Instruments. The Diamond headers are often subtly different, and using the wrong files will lead to obscure errors.

Errors `<errno.h>`

This header contains definitions of macros that relate to the reporting of error conditions. In addition, it provides access to `errno`; users are advised not to access errno via a declaration of their own, as in future versions it

may not simply be the identifier of an object.

Limits <float.h> and <limits.h>

These headers define a number of macros specifying the limits and characteristics of numeric types. Details may be found in section 2.2.4.2 of the ANSI C standard.

Common Definitions <stddef.h>

This header contains definitions of the following types and macros.

NULL	the null pointer constant
offsetof	return the offset of a structure member from the start of the structure
ptrdiff_t	the type of the result of subtracting one pointer from another
size_t	the type of the result of `sizeof` and `offsetof`.
wchar_t	the type of a wide character: see multibyte characters and strings.

`NULL` and `offsetof` are discussed further in the list of functions.

Alt Package <alt.h>

The `<alt.h>` functions allow a program to detect which of a group of input channel becomes ready first. There are two sets of functions. The _nowait set returns a status value if none of the specified channels is ready to communicate. The others wait until a channel becomes ready. There are two ways to specify which channels are to be tested. The _vec functions use an array of pointers to the channels; the others use a variable-length argument list of pointers to channels.

alt_nowait	is any one of a list of channels trying to send?
alt_nowait_vec	is any one of an array of channels trying to send?
alt_wait	await input from any one of a list of channels
alt_wait_vec	await input from any one of an array of channels

Caution

The `<alt.h>` functions may only be used directly on virtual or internal channels. They cannot be used directly on physical channels that run over a hardware link. If you need to use the `<alt.h>` functions on a physical channel, consider adding an intermediate guard thread to echo messages from the physical channel to an internal channel, as shown below.

```
#include <chan.h>

#define SIZE 64

struct map {
   CHAN *phys_chan;
   CHAN *internal_chan;
};

CHAN internal0, internal1;

void guard(void *arg)
{
   struct map *s = (struct map *)arg;
   char buf[SIZE];
   for (;;) {
      chan_in_message(SIZE, buf, s->phys_chan);
```

```
      chan_out_message(SIZE, buf, s->internal_chan);
   }
}

main(int argc, char *argv[], char *envp[],
     CHAN *in[], int ins, CHAN *out[], int outs)
{
   struct map s0, s1;

   s0.phys_chan = in[0];  s0.internal_chan = &internal0;
   s1.phys_chan = in[1];  s1.internal_chan = &internal1;

   chan_init(&internal0);
   chan_init(&internal1);

   thread_new(guard, 1024, &s0);
   thread_new(guard, 1024, &s1);

   for (;;) {
      int i = alt_wait(2, &internal0, &internal1);
      ...
   }
}
```

Here, the `alt_wait` function is applied to two internal channels, internal0 and internal1, but the effect is to alt on the underlying ports, in[0] and in[1], which may be physical channels. Clearly, the guard threads must know the format of the messages expected, in order to echo them properly to the internal channels.

Diagnostics <assert.h>

This header defines the assert macro which assists the programmer in putting run-time diagnostics in a program.

assert	program debugging function

Channels <chan.h>

The functions described here allow programs to access the basic communication facility of the Diamond model, which is to transfer a message across a channel. The header `<chan.h>` defines the following:

- type CHAN representing the channel data type;
- a function to initialise a channel;
- functions to send and receive messages over channels.

The `<chan.h>` functions must always include in their parameter list the address of a channel. This could be the address of a user-declared variable of type CHAN; for example:

```
CHAN mychan;
int value;
chan_init(&mychan);
.
.
.
chan_out_word(value, &mychan);

Alternatively, it could be a port:

int main(int argc, char *argv[], char *envp[],
         CHAN * in_ports[], int ins,
         CHAN *out_ports[], int outs)
```

```
{
   int value;
   .
   .
   .
   chan_out_word(value, out_ports[0]);
}
```

Note that the channels passed in through the port vector arguments of `main` must not be initialised; all others, such as mychan above, must be initialised by calls to `chan_init`.

The functions provided in the `<chan.h>` package are listed below. Messages of any type and size can be transmitted by the two general-purpose functions `chan_in_message` and `chan_out_message`. The word functions are simply convenient shorthands for four-octet messages.

Note that most inter-processor links only support transfers that are multiples of 4 octets.

chan_init	initialise a channel word
chan_in_message	input a message from a channel
chan_in_word	input a 32-bit word from a channel
chan_out_message	output a message to a channel
chan_out_word	output a 32-bit word to a channel

Character Handling `<ctype.h>`

Character Testing Functions

The character testing functions described here are implemented as macros. They return a non-zero value if their argument meets the condition being tested and zero otherwise. The argument is a single integer.

isalnum	determines if the argument is alpha-numeric
isalpha	determines if the argument is alphabetic
iscntrl	determines if the argument is an ASCII control character
isdigit	determines if the argument is a decimal digit
isgraph	determines if the argument is a printing character but not a space
islower	determines if the argument is a lower-case letter
isprint	determines if the argument is a printing character
ispunct	determines if the argument is a punctuation character
isspace	determines if the argument is a space, horizontal or vertical tab, carriage return, form-feed or newline
isupper	determines if the argument is an upper-case letter
isxdigit	determines if the argument is a hexadecimal digit

Character Mapping Functions

tolower	converts upper-case characters to lower case; returns other characters unchanged
toupper	converts lower-case characters to upper case; returns other characters unchanged

Links `<link.h>`

These functions allow a Diamond program to access the physical interprocessor links directly. On certain boards, this may allow you to exchange raw data with external hardware devices. These functions should only

ever reference link that are not mentioned in the configuration file. You should not use the link functions for normal communication between tasks on different processors. Instead, the program should use the channel communication functions. See "links and channels" for more about links and channels.

link_in	input a message from a link
link_in_word	input a 32-bit word from a link
link_out	output a message to a link
link_out_word	output a 32-bit word to a link

Localisation <locale.h>

The localisation facility of ANSI C makes it possible to vary a number of aspects of the run-time library in order to follow local conventions regarding the format of numbers, collating sequences when comparing alphanumeric strings, the format of the time and date, and so on. Currently, Diamond implements the "C" and "" locales only, as required by the ANSI standard.

As well as the following two functions, the header defines a type, lconv, which contains fields relating to the formatting of numbers, and a number of macros which are used to specify aspects of the locale to change or query. For details, see section 4.4 of the standard.

localeconv	return details of numeric formatting conventions of the current locale
setlocale	change or query all or part of the locale

Mathematics <math.h>

The functions described in this section evaluate various standard mathematical functions such as logarithms, sines, cosines etc. The header also defines the macro HUGE_VAL as a double expression that is returned as the result of some of the functions in certain conditions.

When using the mathematical functions, and real arithmetic in general, it is worth bearing in mind that some of the range of C6000 processors do not have hardware floating-point instructions, and that all floating-point operations on them must be carried out by software routines. This has inevitable implications on performance.

Treatment of Error Conditions

Mathematical functions handle errors by returning unusual result values and setting an error code in the external integer variable `errno`.

Trigonometric Functions

The trigonometric functions operate on angles expressed in radians.

acos	returns the arc cosine of the argument
asin	returns the arc sine of the argument
atan	returns the arc tangent of the argument
atan2	returns the arc tangent of the division of the arguments
cos	returns the cosine of the argument
sin	returns the sine of the argument
tan	returns the tangent of the argument

Hyperbolic Functions

cosh	returns the hyperbolic cosine of the argument

sinh	returns the hyperbolic sine of the argument
tanh	returns the hyperbolic tangent of the argument

Exponential and Logarithmic Functions

exp	returns the base e raised to the power of the argument
frexp	splits a floating-point number into a normalised fraction and an integral power of 2
ldexp	multiplies a floating-point number by an integral power of 2
log	returns the natural logarithm of the argument
log10	returns the base-ten logarithm of the argument
modf	breaks the argument into integral and fractional parts

Power Functions

pow	returns the value of the first argument raised to the power of the second argument
sqrt	returns the square root of the argument

Nearest Integer, Absolute Value and Remainder Functions

ceil	returns the smallest value which is equal to or greater than the argument
fabs	returns the absolute value of the floating point argument
floor	returns the largest integer which is less than or equal to the argument
fmod	calculates the floating-point remainder of the division of its arguments

Synchronising Access to the Server <par.h>

In a program in which many execution threads are active, access to the server must be synchronised, so that only one thread may be performing a server operation at one time. If this were not so, for example if two threads attempted to open a file at the same time (using `fopen`), then the two sets of messages to the server would get confused. There are two approaches to resolving this:

1. Make each function in the library thread-safe by automatically claiming and releasing a semaphore around the calls;

2. Require users to do such semaphoring when necessary.

Diamond takes the second approach; the user must semaphore potentially concurrent references to sensitive functions explicitly. There are two reasons for this choice:

1. The need for synchronisation is fairly uncommon, as synchronisation of server accesses between tasks is handled automatically; you only need to protect references within threads of each task.

2. When it is necessary, the synchronisation may need to be performed across a sequence of function calls. For example:

```
printf("vector = [");
for (i=0; i<count; i++) printf(" %d", a[i]);
printf("]\n");
```

   If the synchronisation were performed on each individual call, it would be possible for another thread to get

in and spoil the output.

The required synchronisation is achieved by means of a SEMA variable `par_sema` defined in the header file `<par.h>` , which any thread wishing to use the server must first claim (`sema_wait`) and then release (`sema_signal`) when finished. For example:

```
sema_wait(&par_sema);
   printf("vector = [");
   for (i=0; i<count; i++) printf(" %d", a[i]);
   printf("]\n");
sema_signal(&par_sema);
```

Note that `par_sema` is initialised to 1 before your `main` function is entered; you should not try to initialise it yourself.

Although they do not communicate with the server, the memory allocation functions, `malloc`, `calloc`, `free`, `realloc`, and `memalign` access shared data resources and so must also be protected from each other in the same way with par_sema.

As an alternative to the explicit use of `par_sema`, some of the more common functions used in concurrently executing threads are available in interlocked forms that include these semaphore operations.

par_fprintf	interlocked version of fprintf
par_printf	interlocked version of printf
par_free	interlocked version of free
par_malloc	interlocked version of malloc

For example, `par_printf`(...) is equivalent to:

```
#include <par.h>
sema_wait(&par_sema);
   printf(...);
sema_signal(&par_sema);
```

Functions that need to be protected in this way are marked Server or Heap in the list of run-time library entries.

Semaphores <sema.h>

This group of functions allows a Diamond program to create and manipulate semaphores, which can be used to synchronise the activity of several concurrently executing threads. The header file `<sema.h>` declares a type SEMA, which is used by these functions.

In this version of Diamond, threads are placed in a queue of waiting processes, so that the first thread to start waiting on a semaphore will be the first to be resumed.

sema_init	initialise a semaphore
sema_signal	perform the signal operation on a semaphore
sema_signal_n	perform sema_signal n times
sema_test_wait	check whether waiting on a semaphore would block
sema_wait	perform the wait operation on a semaphore
sema_wait_n	perform sema_wait n times
static_sema_init	Initialisation value for a semaphore (macro)

Events <event.h>

This group of functions allows a Diamond program to create and manipulate events, which can be used to

synchronise the activity of several concurrently executing threads. The header file `<event.h>` declares a type EVENT that is used by these functions.

Events are similar to semaphores, the main difference being that signalling a semaphore can only restart one thread; setting or pulsing an event will restart all of the waiting threads. An event can be in one of three states:

EVENT_NO	The event has not been set and no threads are waiting.
EVENT_YES	The event has been set and so no threads can be waiting.
all other values	The event has not been set and threads are waiting

Programs should not rely on any relationship between the order in which threads start to wait on an event and the order in which they will be resumed.

event_pulse	Indivisibly set and then reset an event
event_init	Initialise an event.
event_set	Set an event.
event_wait	Wait for an event to be set.

Nonlocal Jumps `<setjmp.h>`

These functions enable the programmer to save the current context of the program, and subsequently to return to it. The header defines a type jmp_buf, which is capable of holding all the information necessary to recreate the context.

longjmp	returns to the context saved by setjmp
setjmp	saves the context of the calling function for a subsequent longjmp call

Signal Handling `<signal.h>`

The signal-handling package enables the programmer to create traps for various signals. These events do not arise spontaneously when using Diamond, but have to be raised by the appropriate function call. (This is an ANSI-compliant implementation, by the way; see section 4.7 of the Standard.)

The header defines macros that are used as identifiers for the signals that can be raised, and others which define the action to be taken when a signal is raised; see the synopses in the list of functions.

signal	define way in which a signal is to be handled from now on
raise	raise a signal

Variable Arguments `<stdarg.h>`

A function whose declaration contains an ellipsis "..." may be called with varying numbers of arguments. The facilities described here allow such a function to access its arguments.

The header `<stdarg.h>` defines a type va_list. The user function should declare an object of this type, called the argument pointer, and scan the variable-length argument list with it, using these functions.

Note that it is generally dangerous to attempt to scan over the arguments of a function by incrementing an ordinary pointer as in the following example:

```
void func(int a,...)
{
   // this will not always work
   int b, *p = &a;
```

```
   b = *++p;
}
```

To maintain portability, you should use the `<stdarg.h>` functions instead.

va_start	initialise the argument pointer
va_arg	find the next argument
va_end	finish accessing arguments

Input/Output `<stdio.h>`

The standard I/O functions provide a portable I/O interface for C programs. They are available in the form described here in most implementations of C. They also provide buffering between user programs and files or devices. This means that I/O transfers to or from real files remain efficient even if data is transferred between the file and the user program in small units (e.g., one character at a time). On output, user data is placed in a buffer allocated "behind the scenes" by the standard I/O functions, until the buffer becomes full, at which point the contents of the buffer are written *en masse* to the file. This technique achieves a speed-up because disk devices are optimised for block transfers. The situation for input is similar.

Other standard I/O functions allow random file access and conversion of numeric data between internal (binary) and external (character string) representations.

All of the functions described in this section require the calling program to include the header `<stdio.h>` before they may be called.

Before you can read or write the data in a file, the fopen function must be called to open a path to the file. The name of the file is passed to fopen, which, if the file is accessible, returns a pointer to a structure of type FILE. The calling program must use this file pointer to refer to the file in subsequent I/O operations (`fputc`, for example, requires a file pointer argument to identify the file which is to be written). The FILE type is declared in `<stdio.h>` .

After performing I/O on an open file, the path to the file may be broken by closing the file. Files should be closed when they are no longer in use, since some implementations place a limit on the number of files that may be open at once Files may be opened again after they have been closed. Having more than one pat open to the same file at any point in a program should be avoided, since some implementations may disallow or restrict this. Closing all files explicitly at th e end of a program is, however, unnecessary; this is done automatically by the standard I/O system when a task's `main` function returns or when the exit function is called.

In the following example, a file named fred is opened, some ASCII characters are written out to it and the file is closed. For clarity, no error checking is performed in the example.

```
#include <stdio.h>   // standard I/O declarations

main()
{
   FILE *fp;         // file pointer variable

   fp=fopen("fred",  // file name
            "w");    // open for writing

   fprintf(          // formatted output routine
           fp,       // file pointer (identifies file)
           "Hi!\n"   // text string to be written
          );

   fclose(fp);       // disconnect file
}
```

To simplify writing programs that read one sequential input file, process it and write another sequential output file, most implementations of C provide some means external to a program (e.g., the User Interface) to connect

at run time files or devices other than the default to the standard input and output of a program. This means that programs may be written and tested using the keyboard and screen for standard input and output, then run unchanged using files, without the program itself needing to open the files by name. You can find a description of the mechanism the server uses to redefine the standard I/O streams here.

Stream I/O

The model of I/O supported by the standard I/O package is known as stream I/O.

In the stream I/O model, a file is considered as a sequence of char values. A notional file pointer, maintained by the I/O functions, indicates the character position within the file at which the next character will be read or written. The file pointer is advanced automatically as characters are read or written Allowing the user to position the file pointer supports random file access.

The basic operations provided by the standard I/O package in support of the stream I/O model are therefore "read a character" (`fgetc`), "write a character" (`fputc`), "reposition file pointer" (`fseek`) and "read file pointer" (`ftell`). Other, higher level, operations (e.g., write a string) are built up directly from these primitive operations. Because of this, calls on the character level functions and the higher-level functions may be freely intermixed and characters will still be transferred in the expected order.

Devices such as the keyboard and the screen are included in the stream I/O model: characters may be read or written from them as appropriate (in principle, one at a time), but positioning operations are not supported.

Text and Binary

The basic units in the discussion of stream I/O thus far have been characters. A character is a value of type char or unsigned char. On the C6000 this is held in an octet (or 8-bit byte), which is the smallest addressable unit of memory. Both in the C6000's memory and in a file, one character is held in one octet.

Larger values than those that can be held in a single octet are stored in files as a sequence of octets. Thus, a value of type int will be held as a sequence of four octets. By convention, these will be stored in little-endian order; that is, the octet holding the low-order eight bits of the value is stored in the earliest position in the file, and the octet holding the high-order eight bits in the latest.

Characters may be stored in files as text or binary data. The difference is that text files are organised into lines. From the point of view of the program, the newline character, "\n", separates lines. The program can end a line by outputting a newline character, and on input, the end of a line can be found by comparing the characters being read with the value "\n". Under MS-DOS, these newline characters are inconveniently stored in files as carriage-return line-feed sequences; the conversion between newline and carriage-return line-feed is performed by the server, and is invisible to the program.

Binary files are not divided into lines, and each character is read or written "as is", without conversion.

By default, Diamond reads and writes text files. If you need to process binary data without conversion, you must inform the run-time library that a particular file is to be processed as a binary file. You can do this by using the "binary" specifier "b" in a call to `fopen`. For example:

```
fd = fopen("x.bin", "rb");
```

Notice that the C6000 Diamond implementation of stream I/O is rather different from the implementations on some other processors, notably the TMS320C40, where the smallest addressable unit of memory is a 32-bit word. In particular, these processors require a distinction between packed and unpacked files, which is unnecessary and not implemented on the C6000.

Standard Streams

For convenience, three file pointers are always automatically opened by tasks linked with the full library. These are declared in `<stdio.h>` as follows.

FILE *stdin;	This is the standard input stream. By default, stdin is the keyboard.
FILE *stdout;	This is the standard output stream. By default, stdout is the screen.

FILE *stderr;	This is the standard error stream, used by programs for outputting error messages. It too is normally opened on the screen.

Files processed or created by redirecting the standard input, output and error streams are always text files. You cannot process binary files by redirecting standard input and standard output in this way.

Operations on Complete Files

remove	removes a file from the file system
rename	renames a file
tmpfile	create temporary binary file
tmpnam	generate unique filename

File Access Functions

fclose	closes a file
fflush	writes out any buffered information to the file
fopen	opens a file
freopen	reassigns the address of a FILE structure and reopens the file
setbuf	associates a buffer with an input or output file
setvbuf	determines how stream will be buffered

Formatted Input/Output Functions

fprintf	performs formatted output to a specified file
fscanf	performs formatted input from a file
printf	performs formatted write to standard output
scanf	performs formatted read from standard input
sprintf	performs formatted output to a character string in memory
sscanf	performs formatted input from memory
vfprintf	similar to fprintf, but with a single argument instead of a list of arguments
vprintf	similar to printf, but with a single argument instead of a list of arguments
vsprintf	similar to sprintf, but with a single argument instead of a list of arguments

Character Input/Output Functions

fgetc	returns the next character from a file; generates a true function call
fgets	reads a line from a file; the line is terminated by a NUL character
fputc	writes a single character to a file; generates a true function call
fputs	writes a string to a file
getc	returns the next character from a file; implemented as a macro
getchar	returns the next character from standard input
gets	reads a line from standard input; replacing the newline with a NUL character
putc	writes a single character to a file; implemented as a macro
putchar	writes a single character to standard output
puts	writes a string to standard output; terminates the string with a newline
ungetc	writes a character to a file buffer leaving the file positioned before the character

Direct Input/Output Functions

fread	reads a specified number of items from the file
fwrite	writes the specified number of items to a file

File Positioning Functions

fgetpos	get value of file position indicator
fseek	places the file pointer at a specified character offset relative to the beginning of the file, the end of the file or the current location in the file
fsetpos	set file position indicator
ftell	returns the current character offset from the beginning of the file to the current location within the file
rewind	places you at the beginning of the file

Error Handling Functions

clearerr	resets the error and end of file indicators
feof	tests for end-of-file
ferror	returns a non-zero integer if an error occurs during read or write operations
perror	writes (to stderr) the most recent error encountered

Macros

EOF	Value returned from input functions to indicate End Of File.
NULL	A null pointer value.

General Utilities `<stdlib.h>`

String Conversion Functions

atof	converts an ASCII string to a double value
atoi	converts an ASCII string to an int value
atol	converts an ASCII string to a long value
strtod	converts an ASCII string to a double value
strtol	converts an ASCII string to a long int value
strtoul	converts an ASCII string to an unsigned long int value

Pseudo-Random Sequence Generation Functions

rand	pseudo-random number generator
srand	change seed for rand

Memory Management Functions

Building complex, dynamically-changing data structures requires a special kind of variable storage. Variables

that are static or extern are allocated when a program is written and are therefore not flexible. On the other hand, auto or register variables disappear when the function which created them returns, and do not persist for other functions to access.

The storage class that allowed the most flexible allocation is generally referred to as *heap storage*. In C, heap storage is allocated by calling a library function and remains allocated until it is explicitly released.

calloc	allocates and clears an area of memory
free	deallocates allocated space
malloc	allocates the specified number of contiguous octets of memory
memalign	allocate a memory-aligned area
realloc	changes the size of an allocated area

Communication with the Environment

abort	abnormal program termination (unless trapped)
atexit	set exit handler function
exit	stop program
getenv	access environment variables
system	execute host operating system command

Searching and Sorting Utilities

bsearch	performs a binary search of an array
qsort	sorts an array

Integer Arithmetic Functions

abs	returns the absolute value of the integer argument
div	compute quotient and remainder of an integer division
labs	returns the absolute value of the long int argument
ldiv	compute quotient and remainder of a long int division

Multibyte Character Functions

The ANSI standard allows the character set to include multibyte characters. Multibyte characters may have state-dependent coding, in that a sequence of multibyte characters may include shift characters that alter the interpretation of subsequent characters in the sequence. Such a multibyte string always starts in the same initial shift state.

A multibyte character may be extracted from such a sequence and converted into a single wide character, which is of type `wchar_t` (defined in `<stddef.h>`).

The encoding of multibyte characters depends on the current locale. In Diamond, only the locales "" and "C" are implemented. A multibyte character is always a single char, and `wchar_t` is identical to the char data type. There is no state-dependent coding.

mblen	returns width of a multibyte character
mbtowc	convert a multibyte character to a wide character
wctomb	convert wide character to multibyte character

Multibyte String Functions

As we saw above, in the present version of Diamond multibyte strings and wide character strings both consist of a sequence of one-word characters.

mbstowcs	convert multibyte string to wide character string
wcstombs	convert wide character string to multibyte string

String Handling <string.h>

The C language itself allows the manipulation of single characters. Library functions are provided to allow C programs to process variable-length strings of characters.

Copying Functions

memcpy	copies a given number of bytes from one memory location to another; undefined for overlapping blocks
memmove	"safe" block move
strcpy	copies one string to another
strncpy	copies a maximum number of characters from one string to another

Concatenation Functions

strcat	concatenates two strings
strncat	concatenates two strings up to a maximum number of characters

Comparison Functions

memcmp	compare two blocks of memory
strcmp	performs lexicographic comparison of two ASCII strings
strcoll	compare strings using collating sequence of current locale
strncmp	performs lexicographic comparison of two ASCII strings (up to a maximum number of characters)
strxfrm	transform string using collating sequence of current locale

Search Functions

memchr	locate character in block of memory
strchr	finds a specified character in a string
strcspn	returns the length of the initial part of a string that does not contain specified characters
strpbrk	locate first character from character set
strrchr	find last copy of specified character in string
strspn	returns the length of the initial part of a string that contains specified characters
strstr	locate substring within string
strtok	returns a pointer to the first character of a token

Miscellaneous Functions

memset	overwrites each octet of an object with a given character code
strerror	maps errno codes to strings
strlen	Returns the length of a string

Threads <thread.h>

The functions in this section allow a Diamond program to create new threads of execution within a single task.

You need to decide on a size for the thread's workspace, which is used to hold the thread's stack. This space is needed for several things:

- The auto variables of the function you invoke in the new thread, together with all the other functions it calls;
- A minimum of five words for every level of function calling;
- The stack requirements of any run-time library functions the thread calls. These vary depending on the functions you call; a good rule-of-thumb would be to allow 4K octets for this, unless you call only trivial run-time library functions.

As we have seen, the microkernel arranges for the available time to be shared between the various threads. When a thread is temporarily stopped, data relating to it are stored in its own workspace. You should allow `THREAD_MIN_STACK` bytes for this.

thread_launch	general thread-starting facility
thread_new	simpler shorthand version of thread_launch
thread_priority	return current thread's priority
thread_deschedule	make current thread momentarily unable to execute
thread_set_priority	change the priority of the current thread
thread_set_urgent	make the current thread urgent
thread_stop	stop the current thread
thread_wait	wait for a thread to stop.
THREAD_HANDLE	The type of object returned by the thread creation functions. It may be used by another thread to wait for the termination of the new thread (see `thread_wait`).
THREAD_NOTURG	A macro for the priority of urgent threads (priority 0).
THREAD_URGENT	A macro for the priority of normal threads (priority 1).
THREAD_MIN_STACK	The minimum allowed size, in bytes, of the workspace for a thread.

Thread return codes <errcode.h>

Every Diamond thread maintains information that can be used externally to determine error conditions or other status. The information is held in a structure of the following format:

```
typedef struct {
   UINT32      code;
   const char *text;
   UINT32      v1;
   UINT32      v2;
} errcode_t;
```

The meanings of the fields are not defined; you can use them for any purposes you wish.

The type ERRCODE is defined to be a pointer to an errcode_t structure.

The following functions are provided:

errcode_get	Return a pointer to the errcode_t structure of a thread
errcode_see	Convert an errcode_t structure into a textual format
errcode_set	Set values into the current thread's errcode_t structure.

Date and Time `<time.h>`

The following functions return information about the time.

clock	returns processor time used
time	returns the current calendar time

Note that the ANSI functions difftime, mktime, asctime, ctime, gmtime, localtime and strftime are not implemented in Diamond.

Internal Timer `<timer.h>`

The C6000 has two internal timers: Timer 0 is controlled by the microkernel while timer 1 is available for user programs. User programs cannot access timer 0 directly, but instead use the `<timer.h>` functions which work from an internal clock maintained by the microkernel. The timer currently ticks 1000 times a second, and so has a resolution of 1msec. Note that this applies to threads of all priorities. It is possible that this rate of ticking may be changed in future releases of Diamond. Rather than assume a rate of 1000 ticks a second you should use the value returned by the library function `timer_rate`; this will always return the correct rate. If your module's processor clock does not tick at the rate defined in the standard processor type for that module, you can change the rate using the `CLOCK` attribute of the configurer's PROCESSOR statement. The `CLOCK` attribute can also be used to prevent the kernel from using TIMER 0.

timer_after	indicates whether one clock value is later than another
timer_delay	wait at least a specified number of kernel clock ticks
timer_now	returns the kernel's current clock value
timer_wait	wait until the kernel's clock reaches some value
timer_rate	return number of kernel clock ticks per second (currently 1000)

List of Functions

---

abort [Server]

```
#include <stdlib.h>
void abort(void);
```

abort raises the signal SIGABRT. If this returns (that is, if no signal handler has been nominated for SIGABRT by a call to `signal`) the program is terminated, and the status returned to the host operating system is set to 1 (EXIT_FAILURE). Before termination, all functions registered by atexit will be called, and all the task's files will be closed.

---

abs [Stand-alone]

```
#include <stdlib.h>
```

```
int abs(int arg);
```

abs returns the absolute value of its integer operand. The result returned by abs is not defined if arg is the largest negative integer.

acos [Stand-alone]

```
#include <math.h>
double acos(double x);
```

acos returns the arc cosine in the range [0, #]. If x is outside the range [-1, +1], the value 0.0 is returned, and errno is set to the value EDOM.

alt_nowait [Stand-alone]

```
#include <alt.h>
int alt_nowait(int n, ...);
```

Use alt_nowait to find out which, if any, of a set of channels is attempting to provide data. The function can only be used on internal or virtual channels; it does not work with physical channels.

The parameter n is followed by a series of CHAN * arguments chan0, chan1... which are pointers to the channels to be tested. n is the number of channels to be tested; it must match the actual number of channel pointers passed.

For example:

```
alt_nowait(2, &c0, &c1);
```

alt_nowait returns a non-negative value if another thread has already executed a chan_out_message call (or equivalent) using any one of the specified channels. The returned value will be in the range 0…n-1, indicating which channel (chan0, chan1...) is ready to communicate; a `chan_in_message` call on that channel will then not block. If more than one channel is ready to communicate, one will be arbitrarily chosen.

A negative value is returned if no thread is attempting to send a message on any of the channels tested.

alt_nowait_vec [Stand-alone]

```
#include <alt.h>
int alt_nowait_vec(int n, CHAN *channels[]);
```

Use alt_nowait_vec to find out which, if any, of a set of channels is attempting to provide data. The function can only be used on internal or virtual channels; it does not work with physical channels.

The elements of the array channels are pointers to the channels to be tested. n is the number of elements in the array. Note that the channels themselves need not be in an array.

alt_nowait_vec returns a non-negative value if another thread has already executed a chan_out_message call (or equivalent) using any one of the specified channels. The returned value will be in the range 0…n-1, indicating which channel (channels[0], channels[1]...) is ready to communicate; a `chan_in_message` call on that channel will then not block. If more than one channel is ready to communicate, one will be arbitrarily chosen.

A negative value is returned if no thread is attempting to send a message on any of the channels tested.

alt_wait [Stand-alone]

```
#include <alt.h>
int alt_wait(int n, ...);
```

Use alt_wait to block execution of the calling thread until any one of a set of channels is attempting to provide data. No processor time is consumed while waiting, so alt_wait is to be preferred over a "busy wait" loop that repeatedly calls alt_nowait. The function can only be used on internal or virtual channels; it does not work with physical channels. The parameter n is followed by a series of CHAN * arguments chan0, chan1..., which are pointers to the channels. n is the number of channels; it must match the actual number of channel pointers passed.

For example:

```
alt_wait(2, &c0, &c1);
```

alt_wait will only return when a different thread or task executes a chan_out_message (or variant) call on any one of the specified channels. The returned value will be in the range 0…n-1, indicating which channel (chan0, chan1, …) is ready to communicate; a `chan_in_message` call on that channel will then not block. If more than one channel becomes ready to communicate, one will be arbitrarily chosen.

alt_wait_vec [Stand-alone]

```
#include <alt.h>
int alt_wait_vec(int n, CHAN *channels[]);
```

Use alt_wait_vec to block execution of the calling thread until any one of a group of channels is attempting to provide data. No processor time is consumed while waiting, so alt_wait_vec is to be preferred over a "busy wait" loop that repeatedly calls alt_nowait_vec. The function can only be used on internal or virtual channels; it does not work with physical channels.

channels is an array of pointers to the channels. n is the number of elements in the array. Note that the channels themselves need not be in an array.

alt_wait_vec will only return when a different thread or task executes a chan_out_message (or variant) call on any one of the specified channels. The returned value will be in the range 0…n-1, indicating which channel (channel[0], channel[1], …) is ready to communicate; a `chan_in_message` call on that channel will then not block. If more than one channel becomes ready to communicate, one will be arbitrarily chosen.

asin [Stand-alone]

```
#include <math.h>
double asin(double x);
```

asin returns the arc sine of its argument in the range [-#/2, #/2]. If x is outside the range [-1, +1], the value HUGE_VAL is returned, and errno is set to the value EDOM.

assert [Server]

```
#include <assert.h>
void assert(int expression);
```

If the macro identifier NDEBUG is defined at the point in the source file where `<assert.h>` is included, use of the assert function will have no effect, in particular, expression may not be evaluated. For this reason, the correct functioning of the program must not depend on the execution of any assert

functions. In particular, the expression should have no side-effects.

The assert function puts diagnostics into programs. The expression argument is any scalar expression. When it is executed, if expression is false (that is, evaluates to zero), assert writes a message on the standard error stream and terminates the program. The message gives the filename and line number of the assert call which failed.

No value is returned by assert.

atan [Stand-alone]

```
#include <math.h>
double atan(double x);
```

atan returns the arc tangent of x. The result is in radians.

atan2 [Stand-alone]

```
#include <math.h>
double atan2(double x, double y);
```

atan2 returns the arc tangent of x/y. The result is in radians in the range [-#, #]. If both arguments are zero, the value 0.0 is returned, and errno is set to the value EDOM.

atexit [Stand-alone]

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

The run-time library registers the value of func. The function it points to will be called (with no arguments) at normal program termination, when the main function returns or `exit` is called.

atexit returns 0 if func is registered successfully, otherwise it returns a non-zero value.

There is no practical limit on the number of functions that may be registered. The same function may be registered more than once, and will be called more than once.

atof [Stand-alone]

```
#include <stdlib.h>
double atof(const char *nptr);
```

The string pointed to by nptr is converted to double-precision floating point representation. The format accepted by atof is the same as that accepted by `strtod`; in fact, a call to atof is equivalent to:

```
strtod(nptr, (char **)NULL)
```

atoi [Stand-alone]

```
#include <stdlib.h>
int atoi(const char *nptr);
```

This function converts the string pointed to by nptr to integer representation. The format accepted by atoi is the same as that accepted by `strtol`, with a decimal base; in fact, a call to atoi is equivalent

to:

```
(int)strtol(nptr, (char **)NULL, 10)
```

atol [Stand-alone]

```
#include <stdlib.h>
long atol(const char *nptr);
```

This function converts the string pointed to by nptr to long int representation. The format accepted by atol is the same as that accepted by `strtol`, with a decimal base; in fact, a call to atol is equivalent to:

```
strtol(nptr, (char **)NULL, 10)
```

bsearch [Stand-alone]

```
#include <stdlib.h>
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar) (const void *, const void *));
```

This function searches an array of objects for an element matching a given key. The result of bsearch is a pointer to the array element located by the search; if no match is found, a null pointer is returned.

bsearch is not limited to any particular data type; it is provided with a comparison function which allows it to compare two objects of any arbitrary type used by the program.

The array to be searched starts at base and has nmemb elements, each of which is size chars long. key points to the item to be searched for, which must have the same type as the elements of the array being searched.

The compar argument points to a comparison function which, given pointers to two objects of the same type as those pointed to by key and base, returns a negative integer to indicate the first is "less than" the second, a positive integer to indicate that the first object is "greater than" the second, or 0 to indicate that the two objects are "equal".

Before calling bsearch, the array must be sorted into ascending order with respect to the comparison function pointed to by compar. This operation can often be most easily performed by the `qsort` function that can sort an arbitrary array into order. Like bsearch, it uses a comparison function to determine the ordering to be used.

calloc [Stand-alone] [Heap]

```
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);
```

calloc returns a pointer to enough space for nelem objects of size elsize, or NULL if the request cannot be satisfied. The space returned will be aligned on an 8-byte boundary and will be initialised to zero. Note that elsize must specify the size of the object in octets (eight-bit bytes).

ceil [Stand-alone]

```
#include <math.h>
double ceil(double x);
```

ceil returns as a double value the smallest integer not less than x.

chan_init [Stand-alone]

```
#include <chan.h>
void chan_init(CHAN *chan);
```

This function initialises the channel pointed to by its chan argument, so that it indicates that no threads are currently attempting to communicate through this channel. All channel objects (i.e., all variables declared to be of type CHAN) must be initialised before the first attempt to communicate through them. If this is not done, the first attempt to communicate through the channel will cause the processor to crash.

Note that this function should not be used on any channel connected to a task on another processor. The channel objects bound to a program's input and output ports are already initialised by the calling environment, and should not be initialised again by the program.

chan_in_message [Stand-alone]

```
#include <chan.h>
void chan_in_message(int len, void *b, CHAN *chan);
```

This function reads a message of length len octets (eight-bit bytes) from the channel pointed to by chan into the variable pointed to by b. The value of len must be greater than zero and a multiple of 4. The function will wait if the other end of the channel is not attempting to write.

chan_in_word [Stand-alone]

```
#include <chan.h>
void chan_in_word(int *w, CHAN *chan);
```

This function reads a word-length message from the channel pointed to by chan into the integer variable pointed to by w. The function will wait if the other end of the channel is not attempting to write.

chan_out_message [Stand-alone]

```
#include <chan.h>
void chan_out_message(int len, void *b, CHAN *chan);
```

This function sends a message of length len octets (eight-bit bytes) from the variable pointed to by b to the channel pointed to by chan. The value of len must be greater than zero and a multiple of 4. The function will wait if the other end of the channel is not attempting to read.

chan_out_word [Stand-alone]

```
#include <chan.h>
void chan_out_word(int w, CHAN *chan);
```

This function sends a word-length message consisting of the value w to the channel pointed to by

chan. The function will wait if the other end of the channel is not attempting to read.

clearerr

```
#include <stdio.h>
void clearerr(FILE *stream);
```

clearerr resets any error indication on the named stream.

clock [Server]

```
#include <time.h>
clock_t clock(void);
```

The clock function returns the time elapsed on the host PC since an (unspecified) base time as the best approximation to the processor time used. The type (clock_t) of the value returned by clock is int. The units used are not specified by the ANSI standard. To find the number of seconds that have gone by since the base time, you should divide the result of clock by the value of the macro CLOCKS_PER_SEC, which is 1 in the current implementation. Note that since the base time is not specified, a program will usually call clock at least twice: once at the start of a section of code to be timed, and once at the end. Subtracting the first clock value from the second and then dividing the result by CLOCKS_PER_SEC gives the number of seconds taken by the code section. clock requires communication with the server, and is only accurate to the nearest second. Do not use clock to measure the time taken to execute sections of code on the target processors. Use Diamond's `timer_now` function, which offers millisecond resolution.

cos [Stand-alone]

```
#include <math.h>
double cos(double x);
```

cos returns the cosine of its radian argument. For very large magnitudes of x (|x| greater than about 12000), the function returns 0 and sets errno to ERANGE.

cosh [Stand-alone]

```
#include <math.h>
double cosh(double x);
```

cosh returns the hyperbolic cosine of its argument. If the magnitude of x is too large, HUGE_VAL is returned, and errno is set to the value ERANGE.

div [Stand-alone]

```
#include <stdlib.h>
div_t div(int dividend, int divisor);
```

This function divides dividend by divisor and returns both the quotient and the remainder in a structure of type div_t. This type is defined in `<stdlib.h>` and includes the following fields:

```
int quot;    // contains the quotient
int rem;     // contains the remainder
```

If the division is inexact, the quotient returned is the integer of lesser magnitude that is nearest to the algebraic quotient. If the result cannot be represented, the behaviour of div is undefined.

disconnect_server

```
extern void disconnect_server(int wait);
```

This function is used in the uncommon cases where a running application wishes to disconnect from the host server but continue to execute. The host I/O system will be shut down in the same way as when an application terminates. All file handles (including stdin, stdout, and stderr) will be closed and no further reference should be made to them. You should ensure that your application does not have any connections with the host when this call is made (such as connections established by user-defined service clusters).

The parameter wait has two possible values:

1	Disconnect and wait for a reply from the server. The call will only return when the server has been reconnected and a Notify message sent.
0	Disconnect and return immediately.

The most common use of this function will have wait=1. This will reopen stdin, stdout, and stderr when the Notify message is received from the server, allowing the use of I/O functions such as `printf`.

Typical usage would be as follows:

```
#include <stdio.h>
extern void disconnect_server(int);

main()
{
             printf("Disconnecting the server\n");
             disconnect_server(1);
   printf("Server is connected again\n");
}
```

EOF

```
#include <stdio.h>
```

The value returned from input functions to indicate End Of File.

errcode_get [Stand-alone]

```
#include <errcode.h>
ERRCODE errcode_get(THREAD_HANDLE T);
```

The function returns a point to the given thread's errcode_t structure. If the parameter T is zero, the function returns a pointer to the current thread's structure.

errcode_see [Stand-alone]

```
#include <errcode.h>
ERRCODE errcode_see(ERRCODE e, char *buffer);
```

The values in the structure e are converted into a text string in the given buffer. The text fields of the structure is used as a format for the conversion. If e->text is NULL, the string "%08x %08x" will be used. The final string will be given a prefix of the hexadecimal form of e->code followed by a colon and a space.

For example, assume the following:

```
char buff[128];
   e->code = 0x12345678;
   e->text = "failure. line=%d position=%d";
   e->v1   = 12;
   e->v2   = 99;
```

The call "errcode_see(e, buff)" would put the following string into buff:

```
"12345678: failure. line=12 position=99"
```

errcode_set [Stand-alone]

```
#include <errcode.h>
ERRCODE errcode_set(UINT32 code, const char *text,
                    UINT32 v1, UINT32 v2);
```

The fields of the current thread’s errcode_t structure are set to the values of the parameters with the same names. Before being set, all the fields will have zero values.

errno [Stand-alone]

```
#include <errno.h>
int errno;
```

Some run-time library functions return a simple true/false value to indicate success or failure. For example, fopen returns a pointer to a file descriptor, or a null pointer for failure. Many library functions also set the variable errno to indicate the type of error in more detail. Some functions, like sqrt, have only one possible error type. In the case of sqrt, the value assigned to errno when the argument to sqrt is negative is EDOM—domain error. Some other functions, such as strtol, may provoke several different distinguishable errors; strtol may set errno either to EDOM or to ERANGE—range error. The different values of errno are defined as macros in `<errno.h>` .

The values of errno that a particular function uses are described along with the function. In this version of Diamond, errno may also be assigned a server status code by any function which requires access to file services. For example, a failed call to fopen might set errno to 99—server operation failed.

At entry to a program's main function, errno is zero. A run-time library function that does not detect an error does not guarantee to return errno to this initial state, although it may do so. Thus, unless errno is zeroed immediately before a call to a run-time library function, its value should only be examined if the call is otherwise known to have failed, by examination of the function's return value.

EVENT [Stand-alone]

```
#include <event.h>
```

The type of an event object. Threads are waiting on an EVENT if its value is neither `EVENT_NO` nor

EVENT_YES.

event_init [Stand-alone]

```
#include <event.h>
void event_init(EVENT *event, EVENT *value);
```

event_init will put the given event into a standard state and discard any list of waiting threads. This function must be called before the event can be used for the first time. The second parameter, value, may be one of:

EVENT_YES	the event is left set.
EVENT_NO	the event is left reset

EVENT_NO [Stand-alone]

```
#include <event.h>
```

The value of an EVENT that has not been set and has no threads waiting on it. Threads are waiting on an EVENT if its value is neither EVENT_NO nor EVENT_YES.

event_pulse [Stand-alone]

```
#include <event.h>
void event_pulse(EVENT *event);
```

event_pulse will set the given event and immediately clear (reset) it; the setting and clearing are done indivisibly. All threads waiting on the event will be restarted and the event will be left in the reset state (EVENT_NO) with no waiting threads.

event_set [Stand-alone]

```
#include <event.h>
 void event_set(EVENT *event);
```

event_set sets the given event. All threads waiting on the event will be restarted and the event will be left in the set state (EVENT_YES) with no waiting threads.

event_wait [Stand-alone]

```
#include <event.h>
void event_wait(EVENT *event);
```

event_wait tests the current value of the given event. If the event is set, the calling thread will continue execution. If the thread is not set, the current thread will be suspended and added to the list of threads waiting for the event. All such waiting threads will be restarted when the event is set.

EVENT_YES [Stand-alone]

```
#include <event.h>
```

The value of an EVENT that has been set; there are no threads waiting on it. Threads are waiting on an EVENT if its value is neither EVENT_NO nor EVENT_YES.

exit [Stand-alone]

```
#include <stdlib.h>
void exit(int status);
```

exit is the normal means of terminating program execution. It calls all the functions registered by calls to atexit (in reverse order of registration), and then closes all the task's files. The call to exit never returns.

status is used to tell the operating system about the status of the terminating program. The header `<stdlib.h>` defines two macros so that this may be done in a machine-independent way. If status is zero or EXIT_SUCCESS, the program is terminating successfully. If status is EXIT_FAILURE it is terminating unsuccessfully.

The value of status is given to the host operating system result code: 0 indicates success, and non-zero indicates failure.

An implicit call is made to exit(EXIT_SUCCESS) when the main function returns.

exp [Stand-alone]

```
#include <math.h>
double exp(double x);
```

exp returns e^x. The function returns HUGE_VAL and errno is set to ERANGE if the value of e^x is too large to be represented.

fabs [Stand-alone]

```
#include <math.h>
double fabs(double arg);
```

fabs returns the absolute value of arg.

fclose [Server]

```
#include <stdio.h>
int fclose(FILE *stream);
```

fclose causes any buffers for the specified stream to be emptied, and the file to be closed. Buffers allocated by the standard I/O system are freed.

fclose is called automatically upon calling exit.

fclose returns non-zero if stream is not associated with an output file, or if buffered data cannot be transferred to that file.

feof [Server]

```
#include <stdio.h>
int feof(FILE *stream);
```

feof returns non-zero when end of file is read on the named input stream, otherwise zero. It is implemented as a macro.

ferror [Server]

```
#include <stdio.h>
int ferror(FILE *stream);
```

ferror returns non-zero when an error has occurred reading the named stream, otherwise zero. Unless cleared by clearerr, the error indication lasts until the stream is closed.

ferror is implemented as a macro.

fflush [Server]

```
#include <stdio.h>
int fflush(FILE *stream);
```

fflush causes any buffered data for the named output stream to be written to the file or device associated with that stream. The stream remains open.

fflush is called automatically by close, and when all streams are implicitly closed by `exit`.

EOF is returned if stream is not associated with an output file or if buffered data cannot be transferred to that file.

fgetc [Server]

```
#include <stdio.h>
int fgetc(FILE *stream);
```

fgetc returns the next character from the specified input stream. Successive calls return successive characters from the stream. fgetc is a genuine function, unlike `getc` which is a macro.

EOF is returned at end of file or if a read error occurs.

fgetpos [Server]

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

fgetpos stores the file position in the object pointed to by pos. The type fpos_t is defined in `<stdio.h>` .

If successful, fgetpos returns zero. If it fails, it returns a non-zero value, and sets errno to EBADF for a bad file descriptor, or EINVAL for any other error.

Warning

In the current version of Diamond, fgetpos should only be used with binary files.

fgets [Server]

```
#include <stdio.h>
char *fgets(char *str, int n, FILE *stream);
```

fgets reads a maximum of n-1 characters from the stream and stores them in the string str. Reading stops when a newline has been stored or when end-of-file is encountered. The last character read into str is followed by a NUL character.

fgets normally returns str. If an error occurs, or if end-of-file is encountered before any characters have been read, fgets returns NULL.

Note that fgets behaves differently from `gets` with respect to any terminating newline character: fgets keeps the newline, `gets` deletes it from the string.

---

floor [Stand-alone]

```
#include <math.h>
double floor(double x);
```

floor returns the largest integer not greater than x, expressed as a floating-point value.

---

fmod [Stand-alone]

```
#include <math.h>
double fmod(double x, double y);
```

fmod returns the remainder from x/y.

---

fopen [Server]

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type);
```

fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations; if filename cannot be accessed in the way requested, the function returns NULL.

type is a character string made up of the following parts:

1. A specification of whether the file is to be opened for reading "r", writing "w" or appending "a". This specifier must appear as the first character in the type string.
2. An optional "update" specifier "+". If included, the file is opened for both reading and writing. If omitted, the file is opened in the mode described by the first character of type.
3. An optional specification of whether the file is to be a text file, "t", or a binary file "b". If this specifier is omitted, the file is taken to be a text file. Text and binary files are discussed *here*.

The second and third parts of the type string may appear in any order. For example, "r+b" and "rb+" are equivalent. Here are some examples of possible values for type, along with a description of their interpretation.

"r"	open text file for reading
"rt"	open text file for reading
"rb"	open binary file for reading

"rb+"	open binary file for update
"r+b"	open binary file for update
"w"	truncate and write to, or create, text file
"a"	append to, or create, text file
"ab"	append to, or create, binary file

fopen will fail if the file is to be opened for reading ("r") and it does not exist. For writing ("w") or appending ("a"), the file will be created if it does not exist.

If a file is open to read and write (the type argument includes a "+"), it is not possible to switch directly from reading to writing or vice versa. Instead, there must be a call to `fseek` between them. If this is not done, the results are undefined.

fprintf [Server]

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

The arguments that follow the format argument are output to the specified stream, using `putc`. The format argument controls the way in which the following argument list is converted for output.

The format argument is a character string that contains two types of object: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and output of the next argument.

Each conversion specification is introduced by the character "%". Following the "%" there may be the following, in the given order.

Flags Field. This optional field includes any of the following flags, in any order:

-	The value will be left-justified
+	The value will always start with a sign.
space	If the value does not start with a sign, a space will be placed before it.
#	Use an "alternate form" conversion. The alternate forms depend on the conversion character, as follows:

o	Increase specified precision by one character, so that the leading digit is always 0.
x	Precede non-zero value by "0x"
X, p	Precede non-zero value by "0X"
e,E,f,g,G	Output decimal point even if no digits follow it.
g, G	Do not remove trailing zeroes

0	For the "d", "i", "o", "u", "x", "X", "p", "e", "E", "f", "g", "G" conversion characters, the value is padded with zeroes. If the "-" flag appears as well, "0" is ignored. For the "d", "I", "o", "u", "x", "X", "p" conversion characters, if a precision is specified, the "0" flag is ignored.

Field Width. This optional field is a decimal integer; or an asterisk "*", indicating that the value for the field width is obtained from the next argument, which should be an int.

The converted value is padded on the left (or on the right, if the "-" flag has been specified). If a "0" flag is in force, padding will be with "0" characters; otherwise, it will be with spaces.

Precision. This optional field consists of a "." and either a decimal integer, or an asterisk, "*". If it is an asterisk, the value for the precision is obtained from the next argument, which should be an int. If only a "." is specified, the precision is taken as zero.

The meaning of the precision depends on the conversion character, as follows:

d, I, o, u, x, X	The minimum number of digits.
e, E, f	The number of digits to appear after the decimal point.
g, G	The maximum number of significant digits.
s	The maximum number of characters to be written from a string.

Prefixes. This optional field may contain one of the following:

h	The following "d", "i", "o", "u", "x" or "X" conversion character corresponds to a short int or unsigned short int argument; or, the following n conversion character corresponds to an argument which is a pointer to a short int.
l	The following "d", "i", "o", "u", "x" or "X" conversion character corresponds to a long int or unsigned long int argument; or, the following n conversion character corresponds to an argument which is a pointer to a long int.
L	The following "e", "E", "f", "g" or "G" conversion character corresponds to a long double argument.

Conversion Character. The conversion characters and their meanings are:

d, I	The int argument is converted to decimal notation. The default precision is 1.
o, u, x, X	The unsigned int argument is converted to unsigned octal ("o"), unsigned decimal ("u") or unsigned hexadecimal ("x" or "X"). The default precision is 1. When writing a hexadecimal number, the letters abcdef are used for "x" conversion, and ABCDEF for "X" conversion.
F	The double argument is converted to decimal notation in the form "[–]ddd.ddd" where the number of digits after the decimal point is equal to the precision specification for the argument. The default precision is 6.
e, E	The double argument is converted to decimal notation of the form "[–]d.ddde[±]dd". There is one digit before the decimal point and the number after is equal to the precision specification for the argument; the default precision is 6. The "e" conversion character generates "e" as the exponent character, while "E" generates "E".
g, G	The double argument is output in style "f" or "e". The precision specifies the number of significant digits; the default is 1. Style "e" is only used if the exponent after conversion is less than -4 or greater than or equal to the precision. Style "E" is used in place of "e" if "G" is specified.
C	The int argument is converted to unsigned char and printed.
S	The argument is taken to be a string (character pointer) and characters from the string are printed until a NUL character is reached or until the number of characters indicated by the precision specification is reached; however, if the precision is zero or missing, all characters up to, but not including, a NUL are printed. Note that a NULL pointer is interpreted as an empty string: `fprintf(f, "%s", 0);  // generates no output`
P	The value of the pointer-to-void argument is printed as a hexadecimal number. The default precision is 8.
N	No output is performed. Instead, the number of characters output so far by this call to fprintf is placed in the int variable that the argument points at.
%	Print a "%" character; no argument is converted

In no case does a non-existent or small field width cause truncation of a field. The maximum length for a single converted argument is 512 characters.

fprintf returns the number of characters output, or a negative value if an output error occurred.

fputc [Server]

```
#include <stdio.h>
int fputc(int cval, FILE *stream);
```

fputc appends the character cval to the specified output stream. It returns the character written. fputc, unlike `putc`, is a genuine function rather than a macro.

fputc returns EOF if an error occurs.

fputs [Server]

```
#include <stdio.h>
int fputs(const char *str, FILE *stream);
```

fputs copies the NUL-terminated string str to the specified output stream. The NUL character that terminates the string is not written to the stream.

Note that unlike `puts`, fputs does not append a newline to the output string.

fread [Server]

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

fread reads nitems objects, each size characters in length, from the specified input stream into memory at location ptr. It returns the number of complete items actually read. Zero is returned on error conditions or end of file.

For example, the following code fragment reads ten integer values from the file f into the integer array a:

```
#include <stdio.h>
FILE *f;
int a[10];
…
fread(a, sizeof(int), 10, f);
```

Note that the size argument specifies the size of an item in characters, that is, in octets (eight-bit bytes). The program will always receive size octets for each item, and the same number of octets will be read from the file.

free [Heap] [Stand-alone]

```
#include <stdlib.h>
void free(void *ap);
```

free frees the space pointed to by ap, which will have been obtained originally by a call to `malloc`,

`memalign`, `calloc` or `realloc`. If ap is a null pointer, no action is taken.

It is an error to attempt to free space that was not allocated by a call to `malloc`, `memalign`, `calloc` or `realloc`.

freopen [Server]

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *type, FILE *stream);
```

freopen substitutes the named file filename in place of the open stream. It returns the original value of stream. The original stream is closed.

freopen is typically used to attach the pre-opened constant names stdin, stdout and stderr to specified files.

type is a character string specifying the way in which the file is to be opened. Refer to the description of `fopen` for a full description of the type string.

freopen returns the pointer NULL if filename cannot be accessed.

frexp [Stand-alone]

```
#include <math.h>
double frexp(double value, int *exp);
```

frexp breaks value into its normalised fraction and an integral power of 2. The function returns the fractional part and the integral part is stored in the variable pointed to by exp.

fscanf [Server]

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

fscanf reads characters from the specified input stream, interprets them according to a format string and stores the results in the variables pointed to by the arguments following format.

The format string is regarded as a sequence of directives, which are processed one by one. fscanf tries to match each directive with characters read from the input stream; the way in which this matching is done depends on the directive. If a directive does not match characters from the input stream, we say that a matching error has happened and the character which caused the error is not read; fscanf returns at once.

There are three types of directive:

1. White space of any length will match white space of any length. If the input stream does not have white space at this point, the directive is ignored.
2. A conversion specification, which is a sequence of characters starting with a "%". These are discussed below.
3. Any other character will match the next character of the input stream if they are the same.

A conversion specification consists of the following, in this order:

1. The character "%";

2. An optional character "*", indicating that the converted value is not to be stored;

3. The field width: an optional non-zero integer which specifies the maximum allowable width of the input field;

4. A prefix character, which indicates the type of the associated argument, as shown in the following table:

Prefix	**Conversion characters**		
	"d", "i", "n"	**"o", "u", "x"**	**"e", "f", "g"**
h	short int	unsigned short int	
l	long int	unsigned long int	double
L			long double

   Note that on the C6000, double and long double variables are represented in the same way.

5. One of the conversion specifiers listed below.

Each conversion specification will match a sequence of characters of a particular format, and these characters are read from the input stream. Reading stops when the first character that does not fit into this format is encountered; this character is not read. It is a matching error if no characters are read, that is, if not even one character would fit the assumed format for this specification.

The character sequence that has been read is converted in one of a variety of ways, and the resulting internal value is stored in the variable pointed to by the next argument (unless "* "was included in the specifier). If this variable is not of an appropriate type for the value that has been converted, the effect is undefined.

The following specifiers are recognised:

%	Matches a "%" character. The complete specifier must be "%%". No argument is used.
c	Matches a sequence of characters of the length specified in the field width (1 by default). The argument should be a pointer to an array of characters large enough to accept the string. Note that unlike "s", the "c" specifier does not skip white space; to read the next non-space character, use "%1s".
d	Matches an optionally-signed decimal integer. The argument should be a pointer to an integer
e, f, g	Match a floating-point number with a format such as would be acceptable to the `strtod` function. This means an optionally-signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an "E" or "e" followed by an optionally-signed integer. The argument should be a pointer to a floating-point variable.
i	Matches an optionally-signed integer with a format such as would be acceptable to the `strtol` function with a base value of 0. This means that strings starting with "0x" or "0X" are interpreted as hexadecimal, strings starting with "0" are interpreted as octal, and others as decimal. The argument should be a pointer to an integer.
n	The argument should be a pointer to an integer, and in this integer is written the number of characters read so far by this call to fscanf. The n specifier reads no characters.
o	Matches an optionally-signed octal integer. The argument should be a pointer to an integer.
p	Matches a pointer value of the format output by a p specifier in a `fprintf` function call. The argument should be a pointer to a pointer to void.
s	Matches a character string which includes no white space. The argument should be a pointer to an array of characters large enough to accept the string and a terminating

	NUL character, which will be added.
u	Matches an optionally-signed decimal integer. The argument should be a pointer to an unsigned integer.
x	Matches an optionally-signed hexadecimal integer with a format such as would be acceptable to the `strtoul` function with a base value of 16. This means that the string may, but need not, start with "0x" or "0X". The argument should be a pointer to an integer.
[	This specifier includes all the characters from the "[" up to a later "]". The characters between the brackets are called the scan-set. The specifier matches a sequence of characters all of which are members of the scanset. So, for example, "[aeiou]" would match a sequence of vowels, of any length and in any order. The argument should be a pointer to an array of characters large enough to accept this sequence. If the first character of the scan-set is a "^", then the specifier matches a sequence of characters none of which are members of the scan-set. To enable the scan-set to include a "]", the standard provides that if the scan-set starts with "]" or "^]" this will not end the specifier and another "]" will be needed. In other words, "[])>]" is a valid specifier, defining a scan-set consisting of "]", ")" and ">".

The conversion specifiers "E", "G" and "X"are treated as being equivalent to "e", "g", and "x". In addition, for compatibility purposes only, "F" is accepted as being equivalent to "lf", that is, a floating-point conversion that expects a pointer to double as the argument.

If end-of-file or an input error occurs before any conversion is done, fscanf returns "EOF". Otherwise, it returns the number of input items successfully converted and stored. The specifier n and specifiers including a "*" do not count towards this number.

fseek [Server]

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

fseek sets the file position indicator of the specified stream. The new position is at the signed distance offset characters from a location specified in whence. Three macros are provided for specifying whence:

SEEK_SET	the start of the file
SEEK_CUR	the current file position
SEEK_END	the end of the file

fseek undoes any effects of `ungetc`; that is, characters which have been "pushed back" into a stream by `ungetc` will not subsequently be read. Instead, reading will proceed from the new file position.

fseek returns -1 for improper seeks, or zero for normal completion.

When operating on a text file, fseek's arguments are limited in the following ways:

- offset may only be 0.
- whence may only be SEEK_SET or SEEK_END.

The ANSI standard also allows fseek to be applied to a text file with whence set to SEEK_CUR and offset set to a value previously obtained by applying ftell to the same stream. The current version of Diamond does not support this.

fsetpos [Server]

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

fsetpos sets the file position of the specified stream to the position stored in the object pointed to by pos. This value should have been stored by an earlier call to fgetpos.

If successful, fsetpos returns zero. If it fails, it returns a non-zero value, and sets errno to EBADF for a bad file descriptor, or EINVAL for any other error.

Note

In the current version of Diamond, fsetpos should only be used with binary files.

ftell [Server]

```
#include <stdio.h>
long int ftell(FILE *stream);
```

ftell returns the current value of the offset relative to the beginning of the file associated with the named stream. This offset is measured in characters.

When operating on a text file, ftell may not give an accurate position unless the current position is either at the beginning or the end of the file.

fwrite [Server]

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

fwrite writes nitems objects, each of size characters, from memory at location ptr to the specified output stream. It returns the number of complete items actually written. Zero is returned on error conditions.

For example, the following code fragment writes the contents of the int array a into the file f:

```
#include <stdio.h>
FILE *f;
int a[10];
…
   fwrite(a, sizeof(int), 10, f);
```

Note that the size argument specifies the size of an item in characters, that is, in octets (eight-bit bytes). The program will always send size octets for each item, and the same number of octets will be written from the file.

getc [Server]

```
#include <stdio.h>
int getc(FILE *stream);
```

getc returns the next character from the named input stream. Successive calls on getc return successive characters from the stream. getc is implemented as a macro.

EOF is returned on end of file or when a read error is detected.

getchar [Server]

```
#include <stdio.h>
int getchar(void);
```

getchar() is identical to `getc`(stdin). It returns the next character from the standard input stream stdin. getchar is implemented as a macro.

EOF is returned on end of file or read error conditions.

getenv [Server]

```
#include <stdlib.h>
char *getenv(const char *name);
```

This function asks the host operating system to return the value of the string that is pointed to by name. If name is known to the host operating system, the function returns a pointer to the corresponding global string value; otherwise, a null pointer is returned.

Note that the string value pointed to by getenv will be valid only until the next call on getenv. Subsequent calls on getenv will overwrite the memory used for the original result. If you need to make several calls to getenv, you should therefore copy the value returned by getenv into a local string before making further calls.

Under MS-DOS, name is assumed to be a pointer to a string which is the name of an environment variable, such as PATH, or any of those defined by the SET command.

Note that the names of all environment variables are forced to be upper-case by the command processor. Thus, the result of the following command would be the definition of a variable called FRED set to the value Mixed:

```
# set fred=Mixed
```

gets [Server]

```
#include <stdio.h>
char *gets(char *str);
```

gets reads a string into str from the standard input stream stdin. The input string is terminated by a newline character, which is replaced in str by a NUL character. gets returns its argument as result.

gets returns NULL on end of file or error.

Note that gets works differently to the similarly named `fgets` in its treatment of the terminating newline character: gets deletes the newline, `fgets` keeps it.

_host_in [Stand-alone]

```
// there is no header file
extern void _host_in(size_t bytes, void *buffer);
```

This function is similar to link_in, but gets its data from the next processor nearer to the host (the one that was used to load this processor). The root processor uses this function to read messages from the host. This function is normally only used to communicate with user-written code running on the PC, as it can disrupt communication with the standard host server.

_host_out [Stand-alone]

```
// there is no header file
extern void _host_out(size_t bytes, void *buffer);
```

This function is similar to link_out, but sends its data to the next processor nearer to the host (the one that was used to load this processor). The root processor uses this function to write messages to the host. This function is normally only used to communicate with user-written code running on the PC, as it can disrupt communication with the standard host server.

host_sema_wait [Server]

```
// there is no header file
extern void host_sema_wait(int n);
```

This function waits for the host to signal one of the 10 host event semaphores maintained by the Diamond kernel. Signalling the host semaphore is discussed here.

The parameter "n" identifies the host semaphore on which the current thread wishes to wait, and must be in the range 0 <=n <= 9.

INPUT_PORT [Server]

```
#include <chan.h>
INPUT_PORT(portid, identifier)
```

The INPUT_PORT macro takes the task's input port selected by portid and associates the bound channel with an identifier; this identifier will be a variable of type CHAN. The macro may be used as a declaration in any file used to build a task.

Portid can be:

- an integer constant index into the task's input port vector, selecting a particular port;
- the name of a CONNECT statement in the configuration file, selecting the input port specification for this task.

See also `OUTPUT_PORT`.

For example, given the following extract of a configuration file:

```
TASK A ....
TASK B .... INS=4

CONNECT control A[1] B[2]
```

The main function of B could include the following:

```
INPUT_PORT(control, control_in)
INPUT_PORT(3, data_in);
```

```
main(int argc, char *argv[], char *envp[],
     CHAN *in_ports[], int ins,
     CHAN *out_ports[], int outs)
{
   ...
   chan_in_message(16, b, in_ports[2]);
   chan_in_message(16, b, &control_in);   // same as above

   chan_in_message(12, b, in_ports[3]);
   chan_in_message(12, b, &data_in);     // same as above
```

isalnum [Server]

```
#include <ctype.h>
int isalnum(int cval);
```

Returns a non-zero value if cval is a letter or a digit, 0 otherwise.

isalpha [Server]

```
#include <ctype.h>
int isalpha(int cval);
```

Returns a non-zero value if cval is a letter, 0 otherwise.

iscntrl [Stand-alone]

```
#include <ctype.h>
int iscntrl(int cval);
```

Returns a non-zero value if cval is an ASCII control character (code less than 20_{16}, or code $7F_{16}$), 0 otherwise.

isdigit [Server]

```
#include <ctype.h>
int isdigit(int cval);
```

Returns a non-zero value if cval is one of the digits "0"–"9", 0 otherwise.

isgraph [Stand-alone]

```
#include <ctype.h>
int isgraph(int cval);
```

Returns a non-zero value if cval is a printing character, codes 21_{16}, ("!") to $7E_{16}$ ("~") inclusive. Returns 0 otherwise.

Note that this function treats the character values between 128 and 255 inclusive as non-printable, although most are visible on a PC screen and on some printers.

islower [Stand-alone]

```
#include <ctype.h>
int islower(int cval);
```

Returns a non-zero value if cval is a lower-case letter, 0 otherwise.

isprint [Stand-alone]

```
#include <ctype.h>
int isprint(int cval);
```

Returns a non-zero value if cval is a printing character, codes 20_{16} (space) to $7E_{16}$ ("~") inclusive. Returns 0 otherwise.

Note that this function treats the character values between 128 and 255 inclusive as non-printable, although most are visible on a PC screen and on some printers.

ispunct [Stand-alone]

```
#include <ctype.h>
int ispunct(int cval);
```

Returns a non-zero value if cval is a punctuation character; otherwise 0. A punctuation character is defined as being any printing character (see isgraph) that is not a letter, a digit or a space.

isspace [Stand-alone]

```
#include <ctype.h>
int isspace(int cval);
```

Returns a non-zero value if cval is a space, horizontal or vertical tab, carriage return, newline or form feed character, 0 otherwise.

isupper [Stand-alone]

```
#include <ctype.h>
int isupper(int cval);
```

Returns a non-zero value if cval is an upper-case letter, 0 otherwise.

isxdigit [Stand-alone]

```
#include <ctype.h>
int isxdigit(int cval);
```

Returns a non-zero value if cval is a printing hexadecimal digit, 0 otherwise.

The printing hexadecimal digits are "0" to "9", "a" to "f" and "A" to "F".

_kernel [Stand-alone]

```
extern void *_kernel;
```

This variable is declared in several system header files, including `<c6xkobj.h>` , and gives a pointer to the kernel which is required for certain functions, most importantly, `SC6xKernel_LocateInterface`.

labs [Stand-alone]

```
#include <stdlib.h>
long int labs(long int val);
```

labs returns the absolute value of **`val`**.

If **`val`** is the most negative long int value, LONG_MIN, the result cannot be represented and the value returned is undefined.

ldexp [Stand-alone]

```
#include <math.h>
double ldexp(double x, int exp);
```

ldexp returns the result of x multiplied by the value of two raised to the power exp. If the result is too large, the function returns HUGE_VAL and errno is set to the value of ERANGE.

ldiv [Stand-alone]

```
#include <stdlib.h>
ldiv_t ldiv(long int dividend, long int divisor);
```

This function divides dividend by divisor and returns both the quotient and the remainder in a structure of type div_t. This type is defined in `<stdlib.h>` and includes the following fields:

```
long int quot;    // contains the quotient
long int rem;     // contains the remainder
```

If the division is inexact, the quotient returned is the integer of lesser magnitude which is nearest to the algebraic quotient. If the result cannot be represented, the behaviour of ldiv is undefined.

link_in [Stand-alone]

```
#include <link.h>
void link_in(size_t len, void *b, int link_no);
```

This function reads a message of length len octets (eight-bit bytes) from link link_no into the area of memory pointed to by b. The argument link_no should be an integer specifying one of the processor's links. The function will wait if no data are available on the link.

Caution

This function should be used with care as it can disrupt channel communications. You should normally use the link functions only on links that have been mentioned in

DUMMY WIRE statements. Communication between tasks is more usually carried out by using the `<chan.h>` functions.

link_in_word [Stand-alone]

```
#include <link.h>
void link_in_word(int *w, int link_no);
```

This function reads a word-length message (four eight-bit bytes) from link link_no, and places the value read in the variable pointed to by w. The argument link_no should be an integer specifying one of the processor's links. The function will wait if no data are available on the link.

Caution

This function should be used with care as it can disrupt channel communications. You should normally use the link functions only on links that have been mentioned in DUMMY WIRE statements. Communication between tasks is more usually carried out by using the `<chan.h>` functions.

link_out [Stand-alone]

```
#include <link.h>
void link_out(size_t len, void *b, int link_no);
```

This function sends a message of length len octets (eight-bit bytes) from the area of memory pointed to by b to link link_no. The argument link_no should be an integer specifying one of the processor's links. The function will wait until the link is able to accept the data.

Caution

This function should be used with care as it can disrupt channel communications. You should normally use the link functions only on links that have been mentioned in DUMMY WIRE statements. Communication between tasks is more usually carried out by using the `<chan.h>` functions.

link_out_word [Stand-alone]

```
#include <link.h>
void link_out_word(int w, int link_no);
```

This function sends a word-length message (four eight-bit bytes) consisting of the value w to link link_no. The argument link_no should be an integer specifying one of the processor's links. The function will wait until the link is able to accept the data.

Caution

This function should be used with care as it can disrupt channel communications. You should normally use the link functions only on links that have been mentioned in DUMMY WIRE statements. Communication between tasks is more usually carried out by using the `<chan.h>` functions.

localeconv [Stand-alone]

```
#include <locale.h>
struct lconv *localeconv(void);
```

localeconv returns a pointer to an object of type struct lconv. The format of this structure is described in section 4.4 of the ANSI standard, and the type is defined in `<locale.h>` . The fields of this structure contain information about the way in which numeric values, including monetary values, are output by the run-time library with the current locale.

As the current version of Diamond only supports locales "C" and "", as laid down by the standard, and as both of these have the same characteristics, the values returned for the various members of the lconv structure are always those laid down in 4.4 of the standard.

log [Stand-alone]

```
#include <math.h>
double log(double x);
```

log returns the natural logarithm of x.

If x is negative, log returns HUGE_VAL, and errno is set to the value of EDOM. If x is zero, it returns HUGE_VAL and sets errno to ERANGE.

log10 [Stand-alone]

```
#include <math.h>
double log10(double x);
```

log10 returns the logarithm of x to base 10.

If x is negative, log10 returns HUGE_VAL, and errno is set to the value of EDOM. If x is zero, it returns HUGE_VAL and sets errno to ERANGE.

longjmp [Stand-alone]

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

This function, together with `setjmp`, is useful for dealing with errors encountered in a low-level subroutine of the program. longjmp restores the stack environment saved in its env argument by an earlier call on `setjmp`. This has the effect of resuming execution immediately after that `setjmp` call. `setjmp`'s caller can distinguish between the original return from `setjmp` and the second return caused by longjmp by examining `setjmp`'s return value. This is always 0 for the initial return, and the value of longjmp's val argument for subsequent returns. If val is set to 0, longjmp will change it to a 1 in order to preserve this condition.The function that originally called `setjmp` must not itself have returned before the call to longjmp. All accessible data still have their values as of the time longjmp was called.

malloc [Stand-alone] [Heap]

```
#include <stdlib.h>
void *malloc(size_t nchars);
```

malloc allocates space for an object whose size is specified in octets (eight-bit bytes) by nchars. The function returns a pointer to the start of the allocated space. If the space cannot be allocated, malloc will return a null pointer. The amount of storage available to malloc (the heap) is set by the configurer. See also: calloc, memalign, and realloc. The space returned will be aligned on an 8-byte boundary.

Space allocated by malloc is not initialised by the run-time library, and may contain arbitrary values. If a zeroed area of storage is required, the function `calloc` should be used. Note that `calloc` has two arguments compared to malloc's one. Thus, a call to malloc(n) must be rewritten as calloc(n,1).

If a request for a zero-length block is made, a pointer to a short—but real—block will be returned by malloc. Note, however, that programs intended to be portable to other implementations of C should not make the assumption that this is so; some other implementations return a null pointer instead.

mblen [Stand-alone]

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

If s is a null pointer, mblen returns 0, indicating that, for the current version of Diamond, multibyte character encodings are never state dependent. Otherwise, it returns the width in octets (eight-bit bytes) of the multibyte character pointed to by s. In the current version, this will be 1, unless s is pointing at a null character, in which case it will be 0.

For further details are available here.

mbstowcs [Stand-alone]

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

The multibyte string pointed to by s is converted to a wide character string and stored in the array pointed to by pwcs. Conversion stops when a null character has been converted, or when n elements have been converted. mbstowcs returns the number of elements converted, excluding the terminating zero, if any.

Note that, in the present version of Diamond, multibyte characters and wide characters are both one octet (eight-bit byte) in length and there is no state-dependent encoding, so this function is equivalent to a string copy. All possible element values are valid, so no error return can happen.

Further details are available here.

mbtowc [Stand-alone]

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

If s is a null pointer, mbtowc returns 0, indicating that, for the current version of Diamond, multibyte character encodings are never state dependent. Otherwise, it returns the width in octets (eight-bit bytes) of the multibyte character pointed to by s. In the current version, this will be 1, unless s is pointing at a null character, in which case it will be 0.

In addition, the character pointed to by s will be converted to a wide character and stored in the location pointed to by pwc. In the current version, as both wide and multibyte characters are always 1 octet in length, this is equivalent to copying the character. The argument n specifies the maximum number of octets to be scanned.

Further details are available here.

memalign [Stand-alone]

```
#include <stdlib.h>
void *memalign(size_t alignment, size_t nchars);
```

memalign allocates space for an object whose size is specified in octets (eight-bit bytes) in nchars. The alignment of the space is specified in alignment. Thus, if alignment is 16, the space returned will be aligned on a 16-octet boundary. Note that malloc aligns its result on an 8-byte boundary.

The function returns a pointer to the start of the allocated space. If the space cannot be allocated, memalign returns a null pointer.

Space allocated by memalign is not initialised by the run-time library and may contain arbitrary values. The function `memset` should be used to clear the storage returned by memalign if a zeroed area of storage is required.

If a request for a zero-length block is made, a pointer to a short—but real—block will be returned by memalign. Note, however, that programs intended to be portable to other implementations of C should not make the assumption that this is so; some other implementations return a null pointer instead.

memchr [Stand-alone]

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

The memchr function searches for the value c (converted to an unsigned char) in the memory block starting at s. The memory block is n octets (eight-bit bytes) in length.

The function returns a pointer to the first occurrence of c within the memory block. If the character is not located, a null pointer is returned.

memcmp [Stand-alone]

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

The memcmp function compares the first n octets (eight-bit bytes) of the two objects pointed to by s1 and s2. The result returned will be less than, greater than, or equal to zero according to whether the object pointed to by s1 is less than, greater than, or equal to the object pointed to by s2.

The comparison operation is performed one character at a time on the complete object; a result will be returned when the first difference between the objects is located.

When comparing complex objects, particularly when these were allocated using malloc from the heap, remember to take account of the following:

"Holes" are sometimes introduced into struct or union objects by the compiler to ensure that fields are correctly aligned on appropriate address boundaries. The contents of such ``holes'' are not defined, unless the objects are statically allocated, or explicitly initialised in their entirety by use of memset or calloc.

Character arrays used as string variables may contain string values whose length is less than that of a previous string value held in the same array. The active values may be followed by parts of the previous value, causing problems in a comparison using memcmp.

memcpy [Stand-alone]

```
#include <string.h>
```

```
void *memcpy(void *s1, const void *s2, size_t n);
```

memcpy copies n octets (eight-bit bytes) from the object pointed to by s2 into the object pointed to by s1. memcpy returns the value of s1.

If the two objects pointed to by s1 and s2 overlap, the behaviour of memcpy is undefined. To copy from one object to another which overlaps it, or when it is not known whether the two objects overlap, you can use the `memmove` function instead of memcpy.

memmove [Stand-alone]

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

memmove copies n octets (eight-bit bytes) from the object pointed to by s2 into the object pointed to by s1. memmove returns the value of s1.

If the two objects pointed to by s1 and s2 overlap, memmove will still perform the copy correctly. This is in contrast to memcpy, for which the behaviour would be undefined. If it is known that the objects pointed to by s1 and s2 definitely do not overlap, you can use the faster `memcpy` function instead of memmove.

memset [Stand-alone]

```
#include <string.h>
void *memset(void *ptr, int cval, size_t num);
```

The memset function copies the value of cval (converted to an unsigned char) into each of the first num octets (eight-bit bytes) of the object pointed to by ptr.

The memset function returns the value of ptr.

modf [Stand-alone]

```
#include <math.h>
double modf(double value, double *iptr);
```

modf splits value into its integral and fractional parts. The function returns the signed fractional part and the integral part is pointed to by *iptr.

NULL [Stand-alone]

NULL is a macro defined in `<stddef.h>` , and also in `<locale.h>` , `<stdio.h>` , `<stdlib.h>` and `<string.h>` . It may be used as a null pointer value of any type, such as (char *)0 or (int *)0, for example.

offsetof [Stand-alone]

```
#include <stddef.h>
offsetof(type, member-designator);
```

This macro expands to a constant expression of type size_t, which has the value of the offset in octets (eight-bit bytes) from the beginning of the structure type to member-designator. The member may not be a bit-field.

For example:

```
#include <stddef.h>
size_t ip;
struct s {
   char fill[4];
   int jim;
};
ip = offsetof(struct s, jim);
```

OUTPUT_PORT [Stand-alone]

```
#include <chan.h>
OUTPUT_PORT(portid, identifier)
```

The OUTPUT_PORT macro takes the task's output port selected by portid and associates the bound channel with an identifier; this identifier will be a variable of type CHAN. The macro may be used as a declaration in any file used to build a task.

Portid can be:

- an integer constant index into the task's output port vector, selecting a particular port;
- the name of a CONNECT statement in the configuration file, selecting the output port specification for this task.

See also `INPUT_PORT`.

For example, given the following extract of a configuration file:

```
TASK A .... OUTS=4
TASK B ....

CONNECT control A[1] B[2]
```

The main function of A could include the following:

```
OUTPUT_PORT(control, control_out)
OUTPUT_PORT(3, data_out);

main(int argc, char *argv[], char *envp[],
     CHAN *in_ports[], int ins,
     CHAN *out_ports[], int outs)
{
   ...
   chan_out_message(16, b, out_ports[1]);
   chan_out_message(16, b, &control_out);   // same as above

         chan_out_message(12, b, out_ports[3]);
         chan_out_message(12, b, &data_out);      // same as above
```

par_fprintf [Stand-alone]

```
#include <stdio.h>
#include <par.h>
int par_fprintf(FILE *stream, const char *format, ...);
```

par_fprintf provides access to the function `fprintf` in circumstances where multiple threads are active; access to the standard I/O structures in the run-time library is interlocked through the semaphore par_sema. In other respects, par_fprintf behaves like `fprintf`, with the same arguments. It returns the number of characters output, or a negative value if an output error has occurred.

Do not call this function if the current thread has already claimed `par_sema` by calling `sema_wait`. This could cause the thread to hang indefinitely.

par_free [Stand-alone]

```
#include <par.h>
void par_free(void *ap);
```

par_free provides access to the function `free` in circumstances where multiple threads are active; access to the memory allocation structures in the run-time library is interlocked through the semaphore `par_sema`.

Do not call this function if the current thread has already claimed `par_sema` by calling `sema_wait`. This could cause the thread to hang indefinitely.

par_malloc [Stand-alone]

```
#include <par.h>
void *par_malloc(size_t nwords);
```

par_malloc provides access to the function `malloc` in circumstances where multiple threads are active; access to the memory allocation structures in the run-time library is interlocked through the semaphore `par_sema`.

Do not call this function if the current thread has already claimed `par_sema` by calling `sema_wait`. This could cause the thread to hang indefinitely.

par_printf [Stand-alone]

```
#include <par.h>
int par_printf(const char *format, ...);
```

par_printf provides access to the function `printf` in circumstances where multiple threads are active; access to the standard I/O structures in the run-time library is interlocked through the semaphore `par_sema`. In other respects, par_printf behaves like printf, with the same arguments. It returns the number of characters output, or a negative value if an output error has occurred.

Do not call this function if the current thread has already claimed `par_sema` by calling `sema_wait`. This could cause the thread to hang indefinitely.

par_sema [Stand-alone]

```
#include <par.h>
SEMA par_sema;
```

When a task has more than one thread is running, steps must be taken to ensure that only one thread at

a time makes use of certain run-time library functions. A thread can ensure that this rule is not broken by waiting for the semaphore par_sema before using one of these functions. After finishing with the run-time library, the thread should signal par_sema so that other threads can get access.

par_sema is also used by all the functions of the par package. For this reason, you must not call one of the other par package functions from a thread which has already claimed par_sema as described above.

Note that par_sema is initialised automatically by the run-time library. Do not try to initialise it in your own code.

perror [Stand-alone]

```
#include <stdio.h>
void perror(const char *s);
```

The perror function maps the value in the global variable errno into a textual message, which is printed on the standard error stream stderr. If s is not a null pointer, perror first prints the string pointed to by s followed by a colon and a space. Regardless of the value of s, perror next prints a message corresponding to errno followed by a newline character.

The error messages produced by perror are the same as those that can be obtained by calling the function strerror with errno as argument.

For example, if the current value of errno is EDOM, a call such as perror("myprog") might produce the following output:

```
myprog: domain error
```

pow [Stand-alone]

```
#include <math.h>
double pow(double x, double y);
```

pow returns x^y, the value of x raised to the power of y.

If x is negative and y is not an integral number, pow returns HUGE_VAL and sets errno to the value of EDOM. If x is zero, and y is zero or negative, pow returns HUGE_VAL and sets errno to EDOM. If the result of the function is too large, pow returns HUGE_VAL and sets errno to the value of ERANGE.

printf [Stand-alone]

```
#include <stdio.h>
int printf(const char *format, ...);
```

printf writes output to the standard output stream, stdout. It returns the number of characters that have been output, or a negative value if an output error occurred. The arguments of printf have the same meaning as the fprintf arguments of the same name. See the description of fprintf. A call to printf is equivalent to a call to fprintf as follows:

```
fprintf(stdout, format, ...);
```

You can use printf for debugging non-root nodes. See here for restrictions.

prompt [Stand-alone]

```
#include <stdio.h>
void prompt(const char *string);
```

prompt sets a new value for the string that is used to identify requests for input from stdin. When using the windows server, such input requests bring up a dialog; prompt changes the text used in the title of that dialog. The setting will be used for subsequent input requests until changed by another call to prompt. The maximum length of the string is 63 characters.

ptrdiff_t [Stand-alone]

```
#include <stddef.h>
```

the type of the result of subtracting one pointer from another.

putc [Stand-alone]

```
#include <stdio.h>
int putc(int cval, FILE *stream);
```

putc appends the character cval to the specified output stream. It returns the character written.

EOF is returned on error.

Because it is implemented as a macro, putc treats a stream argument with side-effects improperly. In particular, the following example causes the pointer f to be incremented several times, which is unlikely to be intended:

```
putc(c, *f++);    // DON'T DO THIS
```

putchar [Stand-alone]

```
#include <stdio.h>
int putchar(int cval);
```

putchar(cval) is a macro defined as putc (cval,stdout). The character cval is written to the standard output stream, stdout (normally the screen).

EOF is returned on error.

puts [Stand-alone]

```
#include <stdio.h>
int puts(const char *pstr);
```

puts copies the NUL-terminated string pstr to the standard output stream stdout and appends a newline character. The terminating NUL character is not copied. stdout is normally the screen.

puts appends a newline to the output string but fputs does not.

qsort [Stand-alone]

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar) (const void *, const void *));
```

This function sorts an array of items into ascending order. The array of items is pointed to by base; in the array, there are nmemb elements, with each element in the array being size octets (eight-bit bytes) long.

Note that the type of the elements in the array is completely general: it might be int in a simple program or some complex struct type in a more sophisticated program. The definition of "ascending order" for this arbitrary data type is provided by the function compar that is passed to qsort as a parameter.

The function pointed to by compar takes two arguments, each a pointer to an item of the type that makes up the array pointed to by base. The function returns an integer less than, equal to, or greater than, zero according to whether the object pointed to by its first argument is to be regarded as less than, equal to, or greater than, that pointed to by its second argument. For example, the following function could be used as a comparison function when it is desired to sort an array of doubles into ascending order:

```
static int compare_doubles(const void *v1, const void *v2)
{
   double *d1 = (double *)v1, *d2 = (double *)v2;
   if (*d1 < *d2) return -1;         // less
   if (*d1 > *d2) return  1;         // greater
                  return  0;  // equal
}
```

The corresponding call on qsort might be as follows, assuming an array a of 1000 doubles:

```
qsort(a, 1000, sizeof(double), compare_doubles);
```

Although qsort nominally sorts the array into ascending order, it can sort into any desired order by appropriate choice of the function passed as the compar argument. The array of doubles used in the previous example could have been sorted into descending order of absolute value using the following comparison function:

```
#include <math.h>
int compare_abs_doubles(const void *v1, const void *v2)
{
   double *d1 = (double *)v1, *d2 = (double *)v2;
   if (fabs(*d1) < fabs(*d2)) return  1;         // less => more
   if (fabs(*d1) > fabs(*d2)) return -1;  // more => less
                              return  0;
}
```

Here, fabs has been used to obtain the absolute value of the variables pointed to by each argument. The sign of the return value is opposite from that in the previous example to give the effect of reversing the order in which qsort will sort the array.

Once an array has been sorted into the correct order using qsort, the function bsearch can be used to search for a particular element within the array.

It is not usually advisable to code as follows, for example:

```
return *d1 - *d2;
```

This is because in some circumstances there could be an overflow, resulting in the items being sorted wrongly.

raise [Stand-alone]

```
#include <signal.h>
int raise(int sig);
```

This function raises the signal specified in sig. Macros are provided to represent the allowed values of sig; they are SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV and SIGTERM. The action taken when the signal is raised depends on what action has been specified for that signal by a call to the signal function. If no such call has been made, the default action will be taken; that is, to return to the caller's program. If such a return is made, 0 is returned if the call was successful, or 1 if there was an error.

Note that the allowed signals will only be raised in the current version by means of calls to raise; they will never happen spontaneously.

rand [Stand-alone]

```
#include <stdlib.h>
int rand(void);
```

rand returns successive pseudo-random integers in the range 0 to RAND_MAX, a macro which is defined in `<stdlib.h>` to be 2^{15}-1.

realloc [Stand-alone] [Heap]

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

ptr should point to a heap object previously allocated by a call to malloc, memalign, calloc or realloc.

realloc changes the size of the object pointed to by ptr to the size specified (in octets) by size. The function returns a pointer to the start of the possibly moved object. If the space cannot be allocated, the realloc function returns a null pointer and the object pointed to by ptr is unchanged.

If ptr is a null pointer, the equivalent of a call to malloc is performed, with the specified value of size as the number of octets (eight-bit bytes) required.

realloc may be used on an object previously allocated by memalign, but you should not assume that the resulting pointer meets any particular alignment constraints.

remove [Stand-alone]

```
#include <stdio.h>
int remove(const char *filename);
```

The remove function causes the identified by the string pointed to by filename to be deleted. Subsequent attempts to open the file will fail, unless it is created anew.

Zero is returned if the file has been removed, non-zero if the operation failed.

rename [Stand-alone]

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

The file named old is renamed new. old and new are pointers to NUL-terminated character strings which must be valid host file names.

Zero is returned if the rename operation succeeds, non-zero if it fails.

The host operating system determines whether or not a particular file renaming operation will succeed.

rewind [Stand-alone]

```
#include <stdio.h>
void rewind(FILE *stream);
```

rewind(f) is equivalent to (void) fseek (f,0L,SEEK_SET). It repositions stream to the first character (character 0) of the associated file. It has no effect if the stream is associated with a device rather than a file (e.g., the keyboard or the screen).

scanf [Stand-alone]

```
#include <stdio.h>
int scanf(const char *format, ...);
```

scanf reads input from the standard input stream stdin. It reads characters (via getc), interprets them according to the given format and stores the resulting values in the locations pointed to by the pointer arguments following format.

The exact meaning of the arguments to scanf is the same as that of the arguments of the same name to the function fscanf. In fact, the call

```
scanf(format, ...);
```

is equivalent to

```
fscanf(stdin, format, ...);
```

If an end-of-file or input error occurs before any conversion is done, fscanf returns EOF. Otherwise, it returns the number of input items successfully converted and stored.

sema_init [Stand-alone]

```
#include <sema.h>
void sema_init(SEMA *s, int v);
```

This function initialises the semaphore variable pointed to by s to an initial state in which:

- the queue of threads waiting for the semaphore is empty;
- the value of the semaphore is v.

If a semaphore is left uninitialised, the first sema_signal or sema_wait operation on the semaphore will cause the system to behave unpredictably. The server synchronisation semaphore, par_sema, is initialised for you; do not initialise it.

See also static_sema_init.

sema_signal [Stand-alone]

```
#include <sema.h>
void sema_signal(SEMA *s);
```

If there are threads waiting for the semaphore pointed to by s, the first one to wait will be chosen and made able to execute again. The value of the semaphore under these conditions will always be 0, and will remain unchanged.

Otherwise, when there are no threads waiting for the semaphore pointed to by s, its value will simply be increased by 1.

This function should only be applied to semaphores that have already been initialised (see sema_init).

sema_signal_n [Stand-alone]

```
#include <sema.h>
void sema_signal_n(SEMA *s, int n);
```

This function calls the function sema_signal n times, in sequence. The parameter n may be greater than or equal to zero.

This function should only be applied to semaphores that have already been initialised (see sema_init).

sema_test_wait [Stand-alone]

```
#include <sema.h>
int sema_test_wait(SEMA *s);
```

If the semaphore pointed to by s has a non-zero count value, sema_test_wait decrements the semaphore count and returns a non-zero value.

If the count in the semaphore is zero and a call on sema_wait would have blocked, sema_test_wait will return 0. This allows a task to check to see if waiting on a semaphore would cause its execution to be suspended.

This function should only be applied to semaphores that have already been initialised (see sema_init).

sema_wait [Stand-alone]

```
#include <sema.h>
void sema_wait(SEMA *s);
```

If the value of the semaphore pointed to by s is not zero, its value is reduced by 1.

If the value of the semaphore is 0, it is left unchanged and the current thread is paused and added to the list of threads waiting for the semaphore. It can only be resumed by some future call on sema_signal.

This function should only be applied to semaphores that have already been initialised (see sema_init).

sema_wait_n [Stand-alone]

```
#include <sema.h>
void sema_wait_n(SEMA *s, int n);
```

This function calls the function sema_wait n times, in sequence. The calling thread may be forced to wait at any point in the sequence.

The parameter n may be greater than or equal to zero.

sema_wait_n should only be applied to semaphores that have already been initialised (see sema_init).

_server_terminate_now [Stand-alone]

```
void _server_terminate_now(int exitcode);
```

Use this function when one task in an application must force the host server to stop immediately. Normally, the server waits until all tasks linked with the full run-time library have exited before it stops. When it stops, the server returns exitcode as a status value to the host operating system, as for the exit function. Usually it is better to avoid this function. Alternative ways of stopping an application are discussed in Shutting down cleanly.

Warning

If any other tasks in the application have open files, they will not be properly closed and data may be lost.

setbuf [Stand-alone]

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

setbuf is used after a stream has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer. If buf is the constant pointer NULL, I/O will be performed without any buffering being interposed by the stdio package. A macro BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from malloc upon the first getc or putc on the file, except that output to the standard error stream stderr is normally not buffered.

SetClear [Stand-alone]

```
void SetClear(volatile unsigned int *w,
```

```
                unsigned int Set,
                unsigned int Clear);
```

This function may be used to change bits in a word without allowing an interrupt to occur during the modification. It may be thought of as equivalent to the pseudo code:

```
unsigned int temp;
PreviousState = CurrentInterruptState;
CurrentInterruptState = DISABLED;
temp = *w;
temp = temp | Set;
temp = temp &amp;amp; ~Clear;
*w   = temp;
CurrentInterruptState = PreviousState;
```

It is common for systems to control peripheral devices through memory-mapped registers. SetClear is useful for changing the state of such registers without fear of an interrupt invalidating the change.

setjmp [Stand-alone]

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

This function, together with longjmp, is useful for dealing with errors encountered in a low-level subroutine of the program.

longjmp restores the stack environment saved in its env argument by an earlier call on setjmp. This has the effect of resuming execution immediately after that setjmp call.

setjmp's caller can distinguish between the original return from setjmp and the second return caused by longjmp by examining setjmp's return value. This is always 0 for the initial return, and the value of longjmp}'s val argument for subsequent returns. If val is set to 0, longjmp will change it to a 1 in order to preserve this condition.

The function that originally called setjmp must not itself have returned before the call to longjmp. All accessible data still have their values as of the time longjmp was called.

setlocale [Stand-alone]

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

setlocale enables the user to change or query all or part of the current locale. The part of the locale to affect is specified in category; the following macros are provided to do this: LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC and LC_TIME.

If a locale is specified, the locale for the specified category will be changed to that locale, and the new locale will be returned. If NULL is specified for locale, the current value of the locale for that category will be returned. If the request cannot be honoured, NULL is returned.

In the current version of Diamond, the only recognised locales are "C" and ""; these have the same characteristics, as defined in section 4.4 of the ANSI standard.

setvbuf [Stand-alone]

```
#include <stdio.h>}
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

After the specified stream has been opened, and before any other I/O has been performed on it, the function setvbuf may be used to change its buffering method to use the specified mode. The mode argument determines how the stream should be buffered, and should be one of the following macros, which are defined in `<stdio.h>` :

_IOFBF	The stream is fully buffered;
_IOLBF	The stream is line-buffered;
_IONBF	The stream is unbuffered

If the argument buf is not NULL, the stream may use the buffer it points to instead of one allocated internally by the run-time library. The argument size specifies the size of this buffer.

The function returns zero to indicate success, or a non-zero value if the mode argument is invalid or if the function call cannot be honoured.

signal [Stand-alone]

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

signal defines how a specified signal will be handled from now on. The allowed values for sig are listed in the discussion of raise.

For the second argument, func, you may specify the macros SIG_DFL or SIG_IGN, both of which result in the specified signal being ignored. Alternatively, you can specify the name of a function, called a signal handler, that is to be called when the signal is raised. During the execution of the signal handler, that signal is ignored. Execution of the signal handler may be ended by calling longjmp, exit or abort; or by executing a return, which will resume execution from the point where the signal was raised.

If there is an error in the call of signal, it will return the value of the macro SIG_ERR, and set errno to EINVAL. Otherwise it will return the value of the func argument.

In the present version of Diamond, signals may only be raised by calling the raise function. They do not happen spontaneously.

sin [Stand-alone]

```
#include <math.h>
double sin(double x);
```

sin returns the sine of its radian argument.

For very large magnitudes of x (|x| greater than about 12000), the function returns 0 and sets errno to ERANGE.

sinh [Stand-alone]

```
#include <math.h>
double sinh(double x);
```

sinh returns the hyperbolic sine of its argument.

If the magnitude of x is too large, HUGE_VAL is returned, and errno is set to the value of ERANGE.

sizeof [Stand-alone]

An operator that returns the number of bytes that are associated with an object of the given type. You can specify the type explicitly, as in **`sizeof(struct foo)`** or **`sizeof(int)`**, or implicitly by giving an instance of that type, as in:

```
struct foo MyStruct;
int MyInt;
size_t TheSize;
TheSize = sizeof(MyStruct);
TheSize = sizeof(MyInt);
```

As `sizeof` is an operator, the brackets are optional.

size_t [Stand-alone]

```
#include <stddef.h>
```

the type of the result of `sizeof` and `offsetof`.

sprintf [Stand-alone]

```
#include <stdio.h>
int sprintf(char *pstr, const char *format, ...);
```

sprintf writes formatted output into a character array via a pointer pstr supplied by the caller. It returns the number of characters (octets) written into the array.

The meaning of the format string and the use of the other arguments is as for fprintf.

A NUL character automatically terminates the output string written to pstr. Note that this terminator is not included in the character count returned by sprintf.

sqrt [Stand-alone]

```
#include <math.h>
double sqrt(double x);
```

sqrt returns the square root of x.

sqrt returns HUGE_VAL when x is negative; errno is set to EDOM.

srand [Stand-alone]

```
#include <stdlib.h>
void srand(unsigned int seed);
```

The srand function uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand.

sscanf [Stand-alone]

```
#include <stdio.h>
int sscanf(char *pstr, const char *format, ...);
```

sscanf reads input from the string pstr. It interprets the characters it reads according to the given format string and stores the resulting values in the locations pointed to by the pointer arguments following format.

The exact meaning of the arguments to sscanf is the same as for fscanf.

If the end of the string is found before any conversion is done, sscanf returns EOF. Otherwise, it returns the number of input items successfully converted and stored.

static_sema_init [Stand-alone]

```
#include <sema.h>
SEMA static_sema_init(SEMA S, int Value);
```

static_sema_init constructs a value that can be used to initialise a semaphore, most often when a static SEMA variable is being declared. It corresponds to the dynamic initialisation: `sema_init`(&S, Value);

```
static SEMA GlobalSema = static_sema_init(GlobalSema, 3);
```

strcat [Stand-alone]

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

strcat appends a copy of string s2 to the end of string s1. A pointer to the NUL-terminated result is returned.

strchr [Stand-alone]

```
#include <string.h>
char *strchr(const char *pstr, int cval);
```

strchr locates the first occurrence of cval (converted to a char) in the string pointed to by pstr. The terminating NUL character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

strcmp [Stand-alone]

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether s1 is lexicographically greater than, equal to or less than s2.

strcoll [Stand-alone]

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

strcoll compares its arguments, interpreting both in the light of the LC_COLLATE category of the current locale. It then returns an integer greater than, equal to, or less than 0, depending on whether s1

is lexicographically greater than, equal to or less than s2.

Note that as the current version of Diamond only supports the "C" and "" locales, strcoll is equivalent to a call on `strcmp`.

strcpy [Stand-alone]

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

strcpy copies string s2 to s1, stopping after the NUL character has been moved. s1 is returned. If copying takes place between objects that overlap, the behaviour is undefined.

strcspn [Stand-alone]

```
#include <srting.h>
size_t strcspn(const char *s1, const char *s2);
```

strcspn calculates the length of the initial part of the string pointed to by s1 which consists of characters not from the string pointed to by s2. The terminating NUL character is not considered part of s2. The function returns the length of the part in characters (octets).

strerror [Stand-alone]

```
#include <string.h>
char *strerror(int errnum);
```

This function maps the error number in errnum into a textual error message string, to which it returns a pointer. For example, an errnum argument of EDOM might return a pointer to the string "domain error".

The caller must not modify the string whose address is returned by strerror. In addition, subsequent calls to strerror may overwrite this string with a new error message. Thus, if the result of strerror is not to be used immediately (for example, to be printed out) it should be copied elsewhere until it is needed to avoid being overwritten.

strlen [Stand-alone]

```
#include <string.h>
size_t strlen(const char *pstr);
```

strlen returns the number of non-NUL characters in pstr.

strncat [Stand-alone]

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t num);
```

strncat appends a copy of string s2 to the end of string s1. It copies at most num characters (octets). A pointer to the NUL-terminated result is returned.

strncmp [Stand-alone]

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t num);
```

strncmp compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether s1 is lexicographically greater than, equal to or less than s2. At most num characters (octets) are examined.

strncpy [Stand-alone]

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t num);
```

strncpy copies string s2 to s1. Exactly num characters (octets) are copied: s2 is truncated or NUL-padded as required. The target may not be NUL-terminated if the length of s2 is num or more. s1 is returned.

strpbrk [Stand-alone]

```
#include <string.h>
char *strpbrk(const char *str, const char *cset);
```

The strpbrk function scans the string pointed to by str for the first character in that string which is also contained in the string pointed to by cset. It returns a pointer to this character once located. If the string pointed to by str does not contain any of the characters from the string pointed to by cset then strpbrk returns a null pointer.

The following example shows how strpbrk might be used to scan a string, replacing any vowels with the character "*":

```
char str[] = "this is some example text";
char *p;
while (p = strpbrk(str, "aeiouAEIOU"))  *p = '*';
```

After execution of this code fragment, the array str would contain the string: "th*s *s s*me *x*mpl* t*xt".

strrchr [Stand-alone]

```
#include <string.h>
char *strrchr(char *s, int c);
```

This function locates the last occurrence of c (converted to a char) in the string pointed to by s. It returns a pointer to the located copy of c. If no copy of c can be located in the string, a null pointer is returned.

Note that strrchr treats the NUL character that terminates the string pointed to by s to be part of that string; therefore, a call such as strrchr(s,0) will locate that NUL terminator.

strspn [Stand-alone]

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

strspn calculates the length of the initial part of the string pointed to by s1 which consists of characters from the string pointed to by s2. The function returns the length of the segment in characters (octets).

strstr [Stand-alone]

```
#include <string.h>
char *strstr(const char *str, const char *sub);
```

This function searches within the string pointed to by str for the string pointed to by sub. If the substring cannot be located, a null pointer is returned. Otherwise, strstr returns a pointer to the first occurrence of the substring.

If sub points to an empty string (i.e., just to a NUL character) then strstr returns str.

As an example of the use of strstr, consider the following code fragment:

```
char *str  = "The quick fox jumps.";
char *sub1 = "fox";
char *sub2 = "dog";
char *ans1 = strstr(str, sub1);
char *ans2 = strstr(str, sub2);
```

After the execution of this code fragment, ans1 will contain a pointer to the part of str starting at "fox", i.e., "fox jumps.". On the other hand, str does not contain the substring "dog", so ans2 will contain a null pointer.

strtod [Stand-alone]

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Starting from the place pointed to by nptr, strtod skips over initial white space, then attempts to interpret characters as forming part of a floating-point constant. Conversion stops at the first character that does not fit into the format of the constant.

The format expected is: an optional sign, a sequence of digits optionally including a decimal point, then an optional exponent part, consisting of an "e" or "E", followed by an optionally-signed integer. The value of this constant is returned as the value of the function, and the object pointed to by endptr is set to point to the first character which is not converted (unless endptr is NULL).

If no conversion could be performed or if the string is empty, zero is returned, and the initial value of nptr is stored in the object pointed to by endptr (unless endptr is NULL). If the value is out of range, +HUGE_VAL or –HUGE_VAL, depending on the sign of the value, is returned. If the value causes underflow, zero is returned. In both these cases, errno is set to ERANGE.

strtok [Stand-alone]

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

strtok breaks the string pointed to by s1 into tokens, each of which is delimited by a character from the string pointed to by s2. The first use of strtok must have s1 pointing at a string. Subsequent use can either have s1 pointing at a new string or a null pointer as its first argument. If a null pointer is used, the function starts from the position the last call terminated. s2 can be different for each call. The function returns a pointer to a token or a null pointer if there is no token found.

strtol [Stand-alone]

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

This function converts the initial portion of the string pointed to by nptr to long int representation. First the string is split into three parts: an initial string of white-space characters (which may be empty), a subject string resembling an integer, to be decoded using the radix information specified in base, and a final string which starts at the first character which is not acceptable in the expected format of the subject string, and extends to and includes the terminating NUL character of the input string. Then it attempts to convert the subject string to an integer, and returns the result.

If the value of base is in the range 2–36, the expected form of the subject string is a sequence of digits and letters representing an integer with the radix specified in base. The letters "a" to "z" (or "A" to "Z") are ascribed the values 10..35. Only those characters that are representations of values less than base are allowed. If base has the value 16, the characters "0x" (or "0X") may precede the sequence of letters and digits, but have no effect.

If the value of base is 0, the subject string is treated as hexadecimal (if it starts with "0x" or "0X"), octal (if it starts with "0") or decimal (for any other case). All other values of base are illegal.

Uppercase letters are everywhere equivalent to lowercase ones, and the subject string may start with a plus or minus sign. However, suffixes (like "L" or "U") are not allowed.

The function attempts to detect overflows, and if this happens the value LONG_MAX or LONG_MIN is returned (these are defined in `<limits.h>`), and errno is set to ERANGE.

If the subject string is empty, or base has an illegal value, then zero is returned, errno is set to EDOM, and the object pointed to by endptr is set to the value of nptr (unless endptr is equal to NULL); in all other cases, including overflows, this object is set to the address of the start of the final string. The subject string will be empty if, for example, the input string is empty or contains only white space. Here are some other input strings whose subject strings are empty:

```
"-"  "+"  "0x"  "/"  "- 1"  "0x-5"
```

strtoul [Stand-alone]

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

This function operates in the same way as strtol, except:

- It returns an unsigned long int;
- In the event of an overflow being detected, the value returned is always ULONG_MAX.

strxfrm [Stand-alone]

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

The function transforms the string pointed to by s2 and places the result in the string pointed to by s1. The nature of this transformation is controlled by the LC_COLLATE category of the current locale, and the effect is that two strings that have been transformed in this way can be correctly compared using strcmp. A maximum of n characters (octets is transformed, including the final NUL character; in

any case, transformation stops after a NUL character has been converted. strxfrm returns the number of characters which have been transformed, excluding the NUL character. The s1 argument may be NULL if n is zero.

Note that, as the current version of Diamond only supports the "C" and "" locales, this function simply performs a copy operation.

system [Stand-alone]

```
#include <stdlib.h>
int system(const char *string);
```

string is passed to the host command-line interpreter and executed as if it had been entered as a command. The string argument to the system function should be a valid host command line.

system returns 0 if the server accepts the command; otherwise it returns a non-zero value. Any host return code generated by the command is not passed back to the calling program.

```
system("dir \\mydir\\*.c");
```

Note that it is not normally possible to use a host command that involves the use of the C6000 system. Attempting to do this will normally result in the program calling system being overwritten by the requested program: when the requested command terminates, the server associated with the original program will not be able to communicate with it and will probably appear to "hang".

tan [Stand-alone]

```
#include <math.h>
double tan(double x);
```

tan returns the tangent of its radian argument. You should check the magnitude of the argument to make sure the result is meaningful.

Common error: If you forget to include `<math.h>` , this function will be automatically declared as a function returning int and will give unpredictable results.

tanh [Stand-alone]

```
#include <math.h>
double tanh(double x);
```

tanh returns the hyperbolic tangent of its argument. You should check the magnitude of the argument to make sure the result is meaningful.

Common error: If you forget to include `<math.h>` , this function will be automatically declared as a function returning int and will give unpredictable results.

thread_deschedule [Stand-alone]

```
#include <thread.h>
void thread_deschedule(void);
```

This function causes a thread to become momentarily unable to execute; this will cause it to be descheduled, thus allowing some other thread of the same priority to resume execution in its place.

Eventually, the thread that called thread_deschedule will resume.

Most applications should never need to use thread_deschedule. If you find yourself wanting to use it, first ask yourself:

- Will the usual time-slicing mechanism give a better effect?
- Will the priority mechanism make the call redundant?
- Will the thread be descheduled anyway as the result of a call that may wait (channel operation, `sema_wait`, `timer_delay`, `event_wait`)?

The function can be used by a thread to ensure that it does not "hog" the processor to the detriment of other threads at the same priority. This is particularly important for a thread whose priority is URGENT, as it cannot be pre-empted by a thread of a higher priority nor stopped because it uses up its time-slice.

THREAD_HANDLE [Stand-alone]

```
#include <thread.h>
```

The type of value returned by a call to `thread_new` or `thread_launch`.

thread_launch [Stand-alone]

```
#include <thread.h>
THREAD_HANDLE thread_launch(void (*fn)(void *),
                                                    void  *ws,
                            size_t wssize,
                            int    priority,
                            void  *arg);
```

This function starts a new thread based on the function fn. The new thread will stop either when it executes the function thread_stop, or when fn returns.

The new thread will use the double-aligned area ws as its workspace (stack), which should be wssize octets (eight-bit bytes) long. It may be allocated from the heap using malloc, or it could be a static or auto array variable.

The function returns a (non-zero) handle to the created thread on success. If wssize is less than THREAD_MIN_STACK the function will fail and return 0.

The priority argument defines the priority at which the new thread will run. It is an integer in the range 0–7, where 0 represents the highest priority and 7 the lowest. The header defines the literals THREAD_URGENT and THREAD_NOTURG, corresponding to priority levels 0 and 1 respectively. Normally, new threads should be started at the same priority as the current thread. You can do this by using the function thread_priority (see below) to provide a value for this argument.

The argument arg will be passed to the new thread's function, fn. It could be, for example, a pointer to a simple variable, or to a struct variable containing a number of parameter values. When you pass a pointer to a variable into a thread in this way, you must make sure that the variable does not change before the thread reads it.

By using a cast, it is also possible to pass a value argument to the thread function:

```
thread_launch(func, ws, 1000, 1, (void *)150);
```

The func function could take up its argument like this:

```
void func(void *value)
{
   int param = (int)value;
}
```

In this example, param would take the value 150. Note that this technique will not work with float or double value parameters; they must be passed by reference.

The value returned by the function may be used to refer to the thread (see `thread_wait`).

See also the description of thread_new, which simplifies thread creation by starting a thread at the current priority and allocating the thread's workspace from the heap.

---

thread_new [Stand-alone]

```
#include <thread.h>
THREAD_HANDLE thread_new(void  (*fn)(void *),
                         size_t  wssize,
                         void   *arg);
```

The function fn is started as a new thread, running at the same priority as the current thread, with a workspace of wssize octets (eight-bit bytes). This workspace is taken from the heap (using par_malloc); you can return it to the heap by passing the returned THREAD_HANDLE to par_free once you know that the thread has stopped.

If there is not enough free heap space to create the workspace or if the size of the workspace is less than THREAD_MIN_STACK, thread_new will return NULL.

The return value may also be used to wait for the termination of the thread (see thread_wait).

The argument arg is the argument that will be passed to the new thread's function, fn. This mechanism is the same as the one used by thread_launch.

thread_new is a shorthand way of calling the more general thread creation function thread_launch in the most common circumstances.

A thread that calls thread_new must not have claimed the par_sema semaphore. This is because par_malloc is used to get the workspace from the heap, and par_malloc itself waits for par_sema. So if par_sema has already been claimed, par_malloc will wait for ever and the call to thread_new will never return.

---

thread_priority [Stand-alone]

```
#include <thread.h>
int thread_priority(void);
```

This function returns the priority level of the current thread, which will be an integer in the range 0..7. The literals THREAD_URGENT and THREAD_NOTURG are defined in the header to correspond to levels 0 and 1 respectively.

---

thread_set_priority [Stand-alone]

```
#include <thread.h>
void thread_set_priority(int newpri);
```

This function changes the priority of the current thread to newpri, which must be in the range 0..7.

thread_set_urgent [Stand-alone]

```
#include <thread.h>
int thread_set_urgent(void);
```

This function makes the current thread urgent (priority 0). It returns the priority the thread had previously, allowing the old priority to be restored later by a call to thread_set_priority, if required.

thread_stop [Stand-alone]

```
#include <thread.h>
void thread_stop(void);
```

This function stops the current thread. The current thread is also stopped if the function in which it started returns. Note that returning from the main function of a task linked against the full run-time library will result in the connection with the host being closed. Another thread may wait for the thread to stop by calling the function thread_wait with the thread's handle as argument.

thread_wait [Stand-alone]

```
#include <thread.h>
void thread_wait(THREAD_HANDLE handle);
```

This function waits for another thread to stop. handle must be the value returned when the thread was created (see thread_launch and thread_new).

For example,

```
THREAD_HANDLE h;
h = thread_new(my_thread, 4000, 0);  // start a thread
do_something();
thread_wait(h);                      // wait for it to finish
```

time [Stand-alone]

```
#include <time.h>
time_t time(time_t *timer);
```

The time function determines the current calendar time. The type (time_t) of the value returned by time is int. The value returned is the number of seconds that have elapsed sinc 00:00:00 GMT on 1st January, 1970, according to the host system clock.

If the timer argument is not a null pointer, the result of time is also assigned to the variable pointed to by time. Therefore, the time function can be used in either of two ways, as shown in the following example, where the two statements each assign the current calendar time to the variable t:

```
t = time((time_t *)0);
(void)time(&t);
```

Although the PC software on which time depends attempts to give you the time in GMT, by default it does this on the assumption that you are in the Pacific Standard Time zone. As most people are not in that zone, you will almost certainly need to make the system aware of your actual time zone. Defining the MS-DOS environment variable TZ does this. For example, if you live in Great Britain, you could

define TZ like this:

```
# set tz=GMT
# set tz=GMT1BST              (during Summer Time)
```

timer_after [Stand-alone]

```
#include <timer.h>
int timer_after(int t1, int t2);
```

This function returns non-zero if the kernel clock value t1 is after the kernel clock value t2, and zero otherwise.

The kernel clock, available to threads of all priorities, ticks timer_rate() times per second.

timer_delay [Stand-alone]

```
#include <timer.h>
void timer_delay(int d);
```

This function causes the current thread to wait for at least d ticks of the kernel's clock. This timer is used by threads of all priorities and ticks timer_rate() times per second.

timer_now [Stand-alone]

```
#include <timer.h>
int timer_now(void);
```

This function returns the current value of the kernel's clock. This clock is used by threads of all priorities and ticks `timer_rate`() times per second.

timer_rate [Stand-alone]

```
#include <timer.h>
long int timer_rate(void);
```

This function returns the number of times the kernel's clock ticks per second. This is usually 1000.

Caution

It is likely that the tick rate will be changed in future releases of Diamond. Applications should not assume any particular value for timer_rate.

timer_wait [Stand-alone]

```
#include <timer.h>
void timer_wait(int t);
```

This function causes the current thread to wait until the value of the kernel's clock is t. This clock is used by threads of all priorities and ticks `timer_rate`() times per second. The function will not wait

if the clock is already at or has passed time t. timer_wait(t) is equivalent to `timer_delay`(t – `timer_now`()).

tmpfile [Stand-alone]

```
#include <stdio.h>
FILE *tmpfile(void);
```

This function creates a temporary binary file that will automatically be deleted at the end of the program run. The file is opened for update with wb+ mode.

tmpnam [Stand-alone]

```
#include <stdio.h>
char *tmpnam(char *s);
```

This function generates a unique filename that is not the name of any existing file. Despite the name of the function, a file opened with this name is not automatically deleted at the end of the program run. If the argument s is a null pointer, the filename is generated in an internal buffer; otherwise, s is assumed to be a pointer to an array of at least L_tmpnam chars, and the filename is written there. The value returned is in both cases a pointer to the place where the filename has been written.

You may call tmpnam a maximum of TMP_MAX times, and each time it will generate a different filename. The internal buffer is only guaranteed to remain unchanged until the next call to tmpnam.

In the current implementation, TMP_MAX is 10000_{16}, and L_tmpnam is 9. The form of the generated filenames is tmp$nnnn, where nnnn is a hexadecimal number (using lower-case letters, rather than upper-case.)

tolower [Stand-alone]

```
#include <ctype.h>
int tolower(int cval);
```

If cval is the ASCII code for an upper-case letter, tolower returns the code for the corresponding lower-case letter. Otherwise, the value of cval is returned unchanged.

toupper [Stand-alone]

```
#include <ctype.h>
int toupper(int cval);
```

If cval is the ASCII code for a lower-case letter, toupper returns the code for the corresponding upper-case letter. Otherwise, the value of cval is returned unchanged.

THREAD_MIN_STACK [Stand-alone]

```
#include <thread.h>
```

This macro gives the smallest acceptable size of a thread’s workspace in bytes. The thread creation functions will fail and return NULL if the given workspace size is less than this value. Note that most threads will probably need more space that this.

THREAD_NOTURG [Stand-alone]

A macro giving the priority of normal (not urgent) threads. It has the value 1.

THREAD_URGENT [Stand-alone]

A macro giving the priority of urgent threads. It has the value 0.

ungetc [Stand-alone]

```
#include <stdio.h>
int ungetc(int cval, FILE *stream);
```

ungetc pushes the character cval back on an input stream. That character will be returned by the next getc call on that stream. ungetc returns cval.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

fseek erases all memory of pushed back characters.

ungetc returns EOF if it can't push a character back.

va_arg [Stand-alone]

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

The va_arg macro is used to access the next argument in a variable-length argument list. The parameter ap should be a variable of the type va_list, which is defined in the `<stdarg.h>` header; it must have been initialised by va_start. The macro expands into an expression that has the value of the next argument and the specified type; if this is not in fact the type of the argument, or if there are no more arguments, the behaviour is undefined. For example:

```
#include <stdarg.h>
void ourfunc(char *message, ...);
{
   va_list ap;
   int ival;
   va_start(ap, message);
   ival = va_arg(ap, int);
   va_end(ap);
}
```

On each call of va_arg, the parameter ap is modified to point to the next argument in the list.

va_end [Stand-alone]

```
#include <stdarg.h>
void va_end(va_list ap);
```

When accessing a variable length argument list, a function should call this macro once all the arguments have been processed. This ensures a correct return to the calling function. The parameter ap should be a variable of the type va_list, which is defined in the `<stdarg.h>` header; it must have been initialised by va_start.

va_start [Stand-alone]

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

va_start is called before accessing a variable-length argument list. The parameter ap should be a variable of the type va_list, which is defined in the `<stdarg.h>` header. The parameter parmN should be the parameter in the variable-argument list immediately before the "...".

vfprintf [Stand-alone]

```
#include <stdio.h>
int vfprintf(FILE *stream, char *format, va_list ap);
```

This function corresponds to fprintf, and performs formatted output to the specified stream. As with fprintf, the format argument controls the conversions to be performed. However, the variable argument list has been replaced by the single argument ap, which should be an argument pointer initialised by va_start. For example:

```
#include <stdarg.h>
#include <stdio.h>
void error(char *func_name, char *format, ...)
{
   va_list ap;
   va_start(ap, format);
   fprintf(stderr, "Error in %s: ", func_name);
   vfprintf(stderr, format, ap);
   va_end(ap);
}
```

The function returns the number of characters output, or a negative value if an output error occurred.

vprintf [Stand-alone]

```
#include <stdio.h>
int vprintf(char *format, va_list ap);
```

This function corresponds to printf, and performs formatted output to the standard output stream, stdout. As with printf, the format argument controls the conversions to be performed. However, as with vfprintf, the variable argument list has been replaced by the single argument ap, which should be an argument pointer initialised by va_start.

The function returns the number of characters output, or a negative value if an output error occurred.

vsprintf [Stand-alone]

```
#include <stdio.h>
int vsprintf(char *pstr, char *format, va_list ap);
```

This function corresponds to sprintf, and writes formatted output into a character array via a pointer pstr supplied by the user. As with sprintf, the format argument controls the conversions to be

performed. However, as with vfprintf, the variable argument list has been replaced by the single argument ap, which should be an argument pointer initialised by va_start.

The function returns the number of characters output, or a negative value if an output error occurred.

wchar_t [Stand-alone]

```
#include <stddef.h>
```

The type of a wide character. See <stddef.h>

wcstombs [Stand-alone]

```
#include <stddef.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

The sequence of wide characters pointed to by pwcs is converted to a multibyte string and stored in the array pointed to by s. Conversion stops when a NUL character has been converted, or when the next character stored would exceed the limit of n words. If the two strings overlap, the effect is undefined.

wcstombs returns the number of octets (eight-bit bytes) stored, excluding the NUL character, if any.

Note that, in the present version of Diamond, multibyte characters and wide characters are both one octet in length and there is no state-dependent encoding, so this function is equivalent to a string copy. All possible element values are valid, so no error return can happen.

Further details are available here.

wctomb [Stand-alone]

```
#include <std.lib.h>
int wctomb(char *s, wchar_t wchar);
```

If s is a null pointer, mbtowc returns 0, indicating that, for the current version of Diamond, multibyte character encodings are never state dependent. Otherwise, it returns the width in octets (eight-bit bytes) of the multibyte character corresponding to value of wchar. In the current version, this will always be 1.

In addition, the multibyte character corresponding to the value of wchar will be stored at the location pointed to by s. In the current version, as both wide and multibyte characters are always 1 octet in length, this is equivalent to storing wchar at *s.

Further details are available here.

Chapter 11. Interrupt Handling

Diamond allows you to use interrupts to control the execution of your application. The general procedure for managing interrupts is as follows:

1. If you are using an external interrupt line, reserve it for your exclusive use. See External Interrupt Manager. You need to do this to prevent your application and the Diamond kernel from both using the same external interrupt line. Other interrupt lines do not need to be claimed in this way.

2. Attach a high-level or a low-level handler for the interrupt to the appropriate interrupt source. High-level handlers are used when the interrupt rate is low. If your device generates a high rate of interrupts, you should consider using a low-level interrupt handler.

3. Enable the appropriate interrupt. This will usually require setting a combination of enable bits in the processor's Interrupt Enable Register, IER, and in device-specific registers. When the interrupt occurs, your handler will be invoked and will run with the specific interrupt that has occurred disabled in IER. The interrupt will be re-enabled in IER when the handler returns.

4. Typically, your application will wait for a semaphore or event that the handler will signal to indicate that the interrupt has happened.

Attaching High-level Interrupt Handlers

c6xint_attach_fn [Stand-alone]

```
#include <c6xint.h>
int c6xint_attach_fn(int sel, void (*fn)(void),
                     void *sp, size_t size)
```

This function attaches a C interrupt function, **fn**, to the interrupt selected by sel. Multiple handlers may be attached to the same interrupt, but there is no way to detach a handler from an interrupt.

sel is a CPU interrupt number, not an interrupt source number. Diamond runs with the hardware default mapping of CPU interrupts to interrupt sources, as set out in table 13-4 of the TMS320C6000 Peripherals Reference Guide (SPRU190C, April 1999). At present, there is no way to configure a different mapping. If you are attaching to an external interrupt line (EXTINT4—EXTINT7) you should make sure that you have claimed this interrupt for your own exclusive use. Some Diamond implementations reserve an external interrupt line to manage interprocessor communications. This is described in Reserved Hardware Resources.

fn is a pointer to your handler function. If you choose to write this function in C, it must be declared with the **interrupt** keyword. User-defined handlers are entered with interrupts globally disabled (GIE cleared) and a return address in the IRP register. This matches the requirements of code generated by the C compiler for interrupt functions.

sp points to the start of a stack area of **size** bytes provided by the caller. Before entering the handler, the kernel sets the stack pointer (register B15) to the end of this area, which must remain valid for as long as the interrupt can occur. Therefore it is usually, but not necessarily, allocated from static storage. Using a separate stack for interrupt handlers means that ordinary threads don't have to worry about allocating enough stack space for any interrupt handlers that may be invoked while they are executing.

size is the length, in bytes, of the interrupt handler stack pointed to by **sp**. It must be large enough to accommodate the stack frames of the handler and any functions it calls. Note that if a C interrupt function calls any other functions, including `i_sema_signal` or `i_event_set` (see later), the C compiler will generate code to save all registers on entry to the function and restore them on exit. This can be costly.

c6xint_attach_fn returns a non-zero value if the operation fails or zero if it succeeds. It calls `malloc` to allocate a structure describing the handler and its stack, so it may fail if the heap is full. Also be aware that you may need to protect the call with `par_sema` if c6xint_attach_fn is called while other active threads may be using `malloc`.

For example, to attach a handler to CPU interrupt 15 (interrupt source TINT1 in the default mapping):

```
#include <c6xint.h>
char handler_stack[2048];

void interrupt hanldler(void)
{
   ...
}

main()
{
   c6xint_attach_fn(15, handler,
   handler_stack, sizeof(handler_stack));
   . . .
}
```

Communicating with the Kernel

In general, high-level interrupt handlers must not call Diamond functions, such as those found in `<stdio.h>` and `<chan.h>` . Often however, a handler needs to restart a thread that is waiting for an interrupt. Three special functions that can be called from a high-level interrupt handler (and only from a high-level interrupt handler) are provided for this:

i_sema_signal_n [Stand-alone]

```
#include <c6xint.h>
void i_sema_signal_n(SEMA *s, unsigned int n);
```

This function will signal a semaphore **n** times from an interrupt handler. Up to **n** threads waiting on the semaphore will become eligible to execute once the interrupt handler terminates.

i_sema_signal [Stand-alone]

```
#include <c6xint.h>
void i_sema_signal(SEMA *s);
```

This function will signal a semaphore from an interrupt handler. If a thread is waiting on the semaphore, it will become eligible to execute once the interrupt handler terminates. i_sema_signal is implemented as a macro. It is equivalent to calling `i_sema_signal_n` with the parameter **n** set to one.

i_event_set [Stand-alone]

```
#include <c6xint.h>
void i_event_set(EVENT *e);
```

This function will set an event from an interrupt handler. Any threads waiting for the event will become eligible to execute once the interrupt handler terminates.

Warning

Do not call plain `sema_signal` or `event_set` from an interrupt handler; this will certainly cause your application to fail.

Warning

Always make sure that all semaphores and events used by interrupt handlers have been initialised appropriately before attaching the handler to an interrupt.

When an ordinary thread needs to wait for an interrupt it can simply wait for a semaphore to be signalled or an event to be set. For example:

```
#include <sema.h>
   static SEMA int_sema; // global to handler and waiting thread

   void interrupt my_handler(void)
   {
      i_sema_signal(&int_sema);
   }

   main()
   {
      sema_init(&int_sema, 0);    // BEFORE attaching interrupt
      . . . attach my_handler to interrupt as described above
      for (;;) {
         sema_wait(&dint_sema);   //  Wait for a block of data
         . . . process device data
      }
   }
```

As well as (optionally) signalling semaphores and setting events, handler code can freely read and write the contents of memory. Just remember to declare variables that are shared with the rest of an application as **`volatile`** (this is not necessary for semaphores and events). Don't be tempted into making user threads wait for interrupts by polling flags in memory though: use semaphores or events instead. In general you should avoid polling because it starves other tasks and threads of CPU cycles. Even devices that transfer only a small amount of data per interrupt can be efficiently dealt with if the handler buffers up data in memory and only signals the user thread when a complete block of data has been processed.

The folder `target\c6000\Sundance\examples\interrupts\attachfn` contains example code that handles interrupts using events and semaphores. The examples both set up C6x timer 1 to interrupt once per second. Each second, when the hardware interrupts, the interrupt handler will signal a semaphore or set an event. This wakes the main thread, which prints a message. The message includes the current value of a counter variable updated by the handler.

Enabling and Disabling Global Interrupts

c6xint_off [Stand-alone]

```
#include <c6xint.h>;
unsigned int c6xint_off(void);
```

This function disables interrupts by clearing the GIE bit in the CSR; it returns the original CSR value. It is implemented as a true function to dissuade the compiler from performing unfortunate code reordering.

c6xint_restore [Stand-alone]

```
#include <c6xint.h>
void c6xint_restore(unsigned int old_csr_value);
```

This function restores a previously saved CSR value returned by `c6xint_off`. It is implemented as a true function to dissuade the compiler from performing unfortunate code reordering.

The standard usage of these functions is:

```
unsigned int old;
. . .
old = c6xint_off();    // interrupts off
//
// do something with interrupts disabled
//
c6xint_restore(old);   // restore old CSR/GIE state
```

Do not be tempted to disable and enable interrupts by simply clearing and setting GIE directly in the CSR. Doing so can lead to unexpected behaviour because of code reordering by the compiler and unhelpful processor behaviour involving PGIE.

Note that indiscriminate global disabling of interrupts is likely to have a bad effect on performance.

Interrupt Processing Flow

When an interrupt occurs, the hardware globally disables interrupts (clears GIE) then executes the corresponding Interrupt Service Fetch Packet (ISFP) from the kernel's Interrupt Service Table (IST).

This enters the kernel's interrupt service routine which handles the interrupt as follows:

1. It prepares for interrupt handling by:
   a. disabling the interrupt by clearing the corresponding bit in the Interrupt Enable Register (IER).
   b. pushing two words on the current thread's stack to free up work registers.
   c. saving more work registers and the Interrupt Return Pointer (IRP) in an Interrupt Control Block (ICB). There are 16 of these, one for each CPU interrupt.
   d. popping the two words previously pushed onto the interrupted thread's stack.
   e. globally re-enabling interrupts (setting GIE), allowing for nested interrupt processing.
2. It then invokes each handler function from a chain of handlers associated with the interrupt. Low-level (kernel) handlers are called directly and run with interrupts enabled (to permit nested interrupts), but when calling a high-level handler the kernel:
   a. Saves the stack pointer (register B15) then points it at the top of the handler stack.
   b. Globally disables interrupts (clears GIE).
   c. Sets up IRP so that when the user handler returns, processing of the chain will resume.
   d. Enters the handler, which returns by branching to IRP; this globally re-enables interrupts.
   e. Restores the saved stack pointer.
3. After all handlers have been called, it returns to the kernel which:
   a. globally disables interrupts (clears GIE).
   b. re-enables the active interrupt by setting the corresponding bit in the IER.
   c. restores the previously saved IRP from the ICB, in case there were any nested interrupts.
   d. branches to IRP, which globally re-enables interrupts and resumes execution of the interrupted thread.

Low-level Interrupt Handlers

Applications that need to support devices generating a high rate of interrupts may need to install low-level handlers. These are written in assembler and have the minimum overhead. Typically such handlers will simply acknowledge the interrupt in a device-dependent way, and signal an event to activate a thread to deal with high-level aspects of the device.

Handler structure

A low-level handler is entered when its associated interrupt is taken. The kernel will call the first handler attached to the interrupt, passing in standard parameters. The handler must deal with the interrupt and then pass control on to the next handler for that interrupt. The kernel itself automatically appears as the final handler on the chain, and so regains control once all handlers have executed.

Handlers are identified by a 3-word structure called an Interrupt Control Block (ICB):

```
struct c6xint_ICB {
   struct c6xint_ICB *Next;       // next ICB for this interrupt
   void              *Arg;        // Argument for this handler
   void              *Handler;    // Address of handler code
};
```

Attaching a low-level handler

You attach a low-level handler to an interrupt selector using c6xint_attach_handler:

c6xint_attach_handler [Stand-alone]

```
#include <c6xint.h>
int c6xint_attach_handler(int sel, struct c6xint_ICB *MyICB);
```

Sel selects the interrupt to which the handler must be attached.

MyICB is a pointer to an c6xint_ICB structure that has already had its **`Arg`** and **`Handler`** fields initialised; this structure must continue to exist for the rest of the execution of your application. Once you have attached a handler, you must not alter the values in the ICB and the ICB cannot be removed from the list of handlers for the interrupt. Attaching a handler does not enable the associated interrupt. Note that all handlers attached to an interrupt will be invoked. It is the responsibility of each handler to check that the device it is controlling is the one that has provoked the interrupt.

For example:

```
extern void MyInt12Handler(void);     // the interrupt handler
static struct c6xint_ICB MyInt12ICB;
static int Parameters[3] = {0, 0, 0}; // depends on the handler
...

MyInt12ICB.Arg     = Parameters;
MyInt12ICB.Handler = (void *)MyInt12Handler;
c6xint_attach_handler(12, &MyInt12ICB);

// now enable interrupt 12...
```

When interrupt 12 occurs, the handler will be called and register a2 will contain the address of the array **`Parameters`**.

Taking interrupts

The kernel will perform the following actions when it is activated by an interrupt:

1. It first saves all of the registers used in interrupt handling in a save area reserved for that purpose. Only a small selection of registers is saved.

2. The bit in IER corresponding to the interrupt is cleared.

3. Interrupts are re-enabled by setting GIE=1.

4. It loads the address of the first ICB into register b11 and calls the first handler, as follows:

```
ldw  *+b11[2], b3   ; load address of handler
ldw  *+b11[1], a2   ; load handler's argument
ldw  *+b11[0], b11  ; load address of next handler
nop  2              ; wait for b3 to load
b    b3             ; call the handler
nop  5
```

5. The handler executes and terminates by repeating the above sequence to enter the next handler.

6. The final handler in the list has an ICB that takes control back to the kernel.

7. The kernel restores the registers it saved earlier and either:
   a. enters the scheduler to deal with threads that have been activated by the handlers; or
   b. resumes the interrupted code if no rescheduling is necessary.

Low-level handler context

When a low-level handler is entered, interrupts are enabled (GIE=1), but the particular interrupt that invoked the handler is disabled: the corresponding bit in IER has been cleared.

The following registers may be used by low-level handlers:

a0	work register – may be used freely
a1	work register – may be used freely
a2	handler argument
a3	work register – may be used freely
a4	work register – may be used freely
a11	kernel pointer
b0	work register – may be used freely
b1	work register – may be used freely
b3	work register – may be used freely
b4	work register – may be used freely
b11	address of next ICB

- Register a11 must not be altered.
- Registers a2 and b11 must be set correctly when the current handler passes control on to the next handler in the chain.
- All other processor registers not listed above must be left unaltered.

Accessing the kernel

Low-level handlers may only use two kernel services, `k_event_set` and `k_sema_signal_n`. These functions are accessed through pointers held in the kernel structure addressed by a11:

Word offset from a11	**Contents**
-2	address of k_sema_signal_n
-1	address of k_event_set
0	

k_event_set [Stand-alone]

This service sets an event. Any threads waiting on the event will be made ready to execute. The following registers are used:

Register	value on entry to k_event_set	value on return
a1	unimportant	unknown
a4	address of event word	unknown
b0	contents of event word	unknown
b3	return address	unchanged
b4	unimportant	unknown

For example:

```
      ;this assumes that a2 contains the address of the event to set.
      ldw   *+a11[-1], b3       ; address of k_event set
      nop   4
      b     b3                  ; call k_event set
      ldw   *a2,  b0            ; load event word
      mv    a2,   a4            ; set address of event word
      mvkl  Next, b3            ; return address
      mvkh  Next, b3
      nop
Next:
```

Note

The loading of **b0** and **a4** must have completed by the time control passes to `k_event_set`.

k_sema_signal_n [Stand-alone]

This service signals a semaphore N times. Up to N threads waiting on the semaphore will be made ready to execute. The following registers are used[a]:

Register	value on entry to k_sema_signal_n	value on return
a0	return address	unchanged
a2	contents of first word of semaphore	unknown
a3	address of semaphore	unknown
a4	unimportant	unknown
b0	unimportant	unknown
b1	N	unknown
b3	unimportant	unknown
b4	unimportant	unknown

For example:

```
            ; this assumes that a2 contains the address of the semaphore.
            ldw   *+a11[-2], b3       ; address of k_sema_signal_n
            mv    a2,   a3            ; address of semaphore
            nop   3
            b     b3                  ; call k_event set
            ldw   *a3,  a2            ; load first word of semaphore
            mvkl  Next, a0            ; return address
            mvkh  Next, a0
            mvk   1,    b1            ; N = 1
            nop
      Next:
```

Note

The loading of **a2**, **a3**, and **b1** must have completed by the time control passes to `k_sema_signal_n`.

[a] The use of registers in interrupt handlers has been optimised for the case of `k_event_set` as this is the most common case.

Low-level Interrupt Handler Example

The following is an example of a low-level interrupt handler for interrupt 6. This handler needs to write to a device-specific register at 0x03fe0000 to acknowledge the interrupt. Note that the return from i_event_set goes directly to the next interrupt handler.

```
            .title   "interrupt 6 handler"
            .text
INT_CLEAR   .set  0x03EF0000      ; write 0 to acknowledge
            .def  _interrupt6
            ;-----------------------------------------;
            ; on entry: a11 = kernel pointer          ;
            ;           a2  = &event                  ;
            ;           b11 = address of next icb     ;
            ;           Interrupts are ENABLED        ;
            ; on exit:  b11 = address of next handler ;
            ;           a2  = arg for next handler    ;
            ; free regs:a0, a1, a3, a4, b0, b1, b3, b4 ;
            ;-----------------------------------------;
_interrupt6: ldw    *+a11[-1], b1  ; pick up pointer to i_event_set
            mv     a2,        a4  ; argument for i_event_set
            ldw   *+b11[1],   b3  ; return address (next handler)
            ldw   *+b11[2],   a2  ; next argument
            ldw   *+b11[0],   b11 ; next handler ICB
            b      b1             ; call i_event_set (&return)
            ldw   *a4,        b0  ; contents of event word
            mvkl   INT_CLEAR, a0
            mvkh   INT_CLEAR, a0  ; address INT_CLEAR
            zero   a1
            stw    a1,       *a0  ; acknowledge interrupt
            ;----------------------------;
            ; NOTE: i_event_set corrupts: ;
            ;       b0, a1, a4, b4       ;
            ;----------------------------;
```

This handler could be attached and used as follows:

```
extern void interrupt6(void);     // the interrupt handler
static struct c6xint_ICB ICB6;
static event Event6 = EVENT_NO;   // initialised event
```

```
...
extern cregister unsigned int IER;
ICB6.Arg     = &Event6;
ICB6.Handler = (void *)interrupt6;
c6xint_attach_handler(6, &ICB6);

IER |= (1<<6);                    // enable interrupt 6

for (;;) {
   event_wait(&Event6);
   // the interrupt has occurred – deal with it
   ...
}
```

Chapter 12. External Interrupts

C6000 processors have four external interrupt lines, INT4...INT7, which can be used to control external devices. One or more of these interrupt lines (and DMA engines) may be permanently reserved by Diamond's device drivers, and others may be dynamically assigned during program execution. If you need to use one of these external interrupt lines to handle an external device, your code must first explicitly claim it from a pool maintained by the kernel's external interrupt manager module. Outline code to use INT4 is shown below:

```
#include <ext_int.h>
SC6xExt_Int *xint;
unsigned int got;

xint = SC6xKernel_LocateInterface(_kernel, SIID_SC6xExt_Int);
if (!xint) error();                 // can't locate interface
got = SC6xExt_Int_Claim(xint, 4);   // try to get INT4
if (!got) error();                  // INT4 already in use
...do things that use INT4...
SC6xExt_Int_Release(xint, 4);       // return it to the kernel
```

SC6xExt_Int_Claim [Stand-alone]

```
#include <ext_int.h>
unsigned int SC6xExt_Int_Claim(SC6xExt_Int *x,
                               unsigned int wanted);
```

This function tries to allocate the requested interrupt line, "**wanted**" . If it succeeds, it returns a non-zero value. If it fails, it returns zero. The **(SC6xExt_Int *)** argument **x** must be a pointer to the kernel's external interrupt manager interface as returned by a call to SC6xKernel_LocateInterface with a second argument of **SIID_SC6xExt_Int**.

SC6xExt_Int_Claim_Any [Stand-alone]

```
#include <ext_int.h>
unsigned int SC6xExt_Int_Claim_Any(SC6xExt_Int *x);
```

This function is similar to SC6xExt_Int_Claim, except that instead of being told which interrupt line to allocate, it finds any free interrupt line. If it succeeds, it returns the interrupt number allocated. If it fails, it returns zero.

SC6xExt_Int_Release [Stand-alone]

```
#include <ext_int.h>
void SC6xExt_Int_Release(SC6xExt_Int *x, unsigned int which);
```

This function is used to return an interrupt line to the available pool. The parameter **which** identifies an interrupt line that had previously been claimed using SC6xExt_Int_Claim or SC6xExt_Int_Claim_Any.

Chapter 13. DMA

This chapter deals with the DMA channels provided on the C620x and C670x processors. Refer to the following chapter for information on EDMA channels.

C6000 processors have a limited number of DMA channels, usually far fewer than the number of threads that might want to use DMA. The Diamond kernel manages all of the available channels and dynamically allocates them to concurrently active inter-processor `<chan.h>` and `<link.h>` calls. User code that wants to make direct use of the channels must claim them from the kernel, complete the DMA operation, and return the channels to the kernel.

The following code fragment demonstrates this by using DMA1 to copy count 32-bit words from memory at here to memory at there:

```
#include <dma.h> ❶

#define START_DMA (DMA_P_SRC_DIR(1) | DMA_P_DST_DIR(1) |  \
                   DMA_P_TCINT(1)   | DMA_P_START(1))

static SC6xDMA *dmaI;

void get_dmaI(void)
{
   if (!dmaI) {                          // only done once
      dmaI = SC6xKernel_LocateInterface(_kernel, SIID_SC6xDMA); ❷
      if (!dmaI) error();                // no interface
   }
}

DMA_REG        *dma;
SC6xDMAChannel *channel;
unsigned int   *here, *there, count;
…
get_dmaI();                              // init dmaI pointer
dma = SC6xDMA_Claim(dmaI, 1, &channel);// claim DMA1  ❸
if (!dma) error();                       // failed to get it?
dma->sec_control = DMA_S_BLOCK_IE(1);  // enable block int
dma->counter     = count;              // measured in words
dma->dst_address = here;
dma->src_address = there;
SC6xDMAChannel_Operation(channel, START_DMA); ❹
SC6xDMAChannel_Release(channel);       // finished with channel ❺
```

There are several things to notice about this code:

❶ This declares the kernel functions used in the rest of the code and creates a reference, `_kernel`, to kernel data structures. It also contains a typedef for a structure type, **`DMA_REG`**, which can be used to access the DMA channel hardware registers, plus definitions for the global DMA registers and the bit fields within the primary and secondary DMA control registers. The **`START_DMA`** macro in the example is based on these bit field definitions.

❷ **`get_dmaI()`** finds and returns a pointer to the kernel's DMA manager interface. This pointer is required in order to claim a DMA channel. Production code would be unlikely to find the interface repeatedly.

❸ **`SC6xDMA_Claim()`** actually claims the channel (DMA1). It returns a pointer to the corresponding DMA hardware registers, or NULL if the requested DMA engine cannot be allocated (because it is already claimed by another thread, or by the kernel for an inter-processor link communication). Using this pointer, the example code then fills in the required values in the DMA1 secondary control, counter, destination and source address registers, as members of the **`DMA_REG`** structure. If SC6xDMA_Claim() succeeds, it returns an SC6xDMAChannel pointer via its final argument. This pointer refers to a software structure in the kernel that describes the allocated DMA channel.

❹ **`SC6xDMAChannel_Operation()`** is one of the functions that can be applied to such a DMA channel pointer. It sets the primary control register of the DMA channel referred to by its first argument to the value given in the second argument, which should set the **`start`** bit (and, in more complex examples, any

required synchronisation of the DMA transfer with an interrupt source). The function then suspends the calling thread until the DMA channel interrupts at the end of the block (as specified by the setting of the secondary control register). While the thread is suspended and the DMA operation is executing, other threads can continue to execute on the CPU. The kernel will catch the DMA completion interrupt, resume the suspended thread and return control to the caller. In this example the DMA channel is no longer required and is released for use elsewhere.

❺ **`SC6xDMAChannel_Release()`** informs the kernel that a previously-claimed DMA channel is no longer required and can be returned to the kernel's pool of free channels.

There is no obligation to use **`SC6xDMAChannel_Operation()`**; it is provided to make handling DMA interrupts easier—but you are free to wait for DMA completion either by polling (not recommended) or by manually installing an interrupt service routine for the DMA interrupt, using the `<c6xint.h>` functions described later. The only mandatory step is to claim the DMA channel before attempting to touch the corresponding hardware. Failure to do so will result in mysterious hangs when your code clashes with a concurrent inter-processor `<chan.h>` or `<link.h>` call in some other task or thread, and the kernel then decides to service that call using the same DMA channel that you are already using for something else.

SC6xDMA Functions

This group of functions allows you to claim DMA channels from the kernel's pool for your own use. You must use one of these functions before touching the DMA hardware. Each of the functions returns a **`(DMA_REG *)`** pointer to the allocated DMA channel's hardware registers, or NULL if the requested DMA channel cannot be allocated. See `<DMA.H>` in the Diamond installation folder for the names and types of the **`DMA_REG`** structure's members.

SC6xDMA_Claim [Stand-alone]

```
#include <dma.h>
DMA_REG *SC6xDMA_Claim(SC6xDMA *dmaI, int n, SC6xDMAChannel **c);
```

This function attempts to allocate DMA channel number **`n`** from the kernel's pool of free channels. If it succeeds, it returns a pointer to the requested channel's hardware control registers. If it fails, it returns NULL.

`dmaI` must be a pointer to the kernel's DMA manager interface, as returned by a call to `SC6xKernel_LocateInterface` with a second argument of **`SIID_SC6xDMA`**.

`n` is the requested DMA channel number. It must be in the range 0–3.

`c` is a pointer to an **`(SC6xDMAChannel *)`** output variable passed by the caller. The function modifies this variable to point to a DMA channel descriptor in the kernel. This pointer is used to refer to the allocated DMA channel when calling the functions described below.

SC6xDMA_ClaimWait [Stand-alone]

```
#include <dma.h>
DMA_REG *SC6xDMA_ClaimWait(SC6xDMA         *dmaI,
                           int              n,
                           SC6xDMAChannel **c);
```

This function attempts to allocate DMA channel number **`n`** from the kernel's pool of free channels. If the requested channel is unavailable, the calling thread will be suspended (indefinitely) until it does become available. Therefore this function will never fail simply because the requested channel is not available. When it succeeds, it returns a pointer to the requested channel's hardware control registers. If it fails for some other reason, it returns NULL. The arguments are the same as for `SC6xDMA_Claim` above.

SC6xDMA_ClaimAny [Stand-alone]

```
#include <dma.h>
DMA_REG *SC6xDMA_ClaimAny(SC6xDMA *dmaI,
                          int oper, SC6xDMAChannel **c);
```

This function is similar to SC6xDMA_Claim, except that instead of requesting a particular channel number, the call returns a pointer to any DMA channel that happens to be free. If no channel is presently free, or if the call fails for some other reason, it returns NULL.

dmaI and **c** have the same meanings as for SC6xDMA_Claim.

oper must be either **READ_OPERATION** or **WRITE_OPERATION**, two values defined as macros by <DMA.H> . This argument is a hint to the function about whether the caller is more likely to use the allocated DMA channel for reading or writing. The kernel uses this information when there is more than one free DMA channel. If **WRITE_OPERATION** is specified, it will allocate the lowest-numbered (highest priority) DMA channel currently available. A FIFO in the DMA engine buffers the highest priority active DMA channel. This has most benefit when writing. If **READ_OPERATION** is specified, the kernel will allocated the highest-numbered (lowest priority) DMA channel available.

SC6xDMA_ClaimAnyWait [Stand-alone]

```
#include <dma.h>
DMA_REG *SC6xDMA_ClaimAnyWait(SC6xDMA *dmaI,
                              int oper, SC6xDMAChannel **c);
```

This function is similar to SC6xDMA_ClaimAny, except that if no channel is presently free, the calling thread will be suspended (indefinitely) until a channel does become free. Therefore this function will never fail simply because a channel is not available. When it succeeds, it returns a pointer to the allocated channel's hardware control registers. If it fails for some other reason, it returns NULL. The arguments are the same as for SC6xDMA_ClaimAny above.

SC6xDMAChannel Functions

These functions all operate on one of the **SC6xDMAChannel** pointers returned by the "claim" functions described above. Note that functions dealing with external devices do not set the various enables that are necessary to allow DMA synchronisation or CPU interrupts. Refer to your C6000 module's hardware documentation for a description of enabling events and interrupts for particular devices. DMA interrupts are automatically managed for you.

SC6xDMAChannel_Release [Stand-alone]

```
#include <dma.h>
void SC6xDMAChannel_Release(SC6xDMAChannel *channel);
```

channel must be a pointer returned by one of the claim functions listed above. The DMA channel described by this pointer is released to the free pool.

SC6xDMAChannel_ResetEvent [Stand-alone]

```
#include <dma.h>
void SC6xDMAChannel_ResetEvent(SC6xDMAChannel *channel);
```

Each **SC6xDMAChannel** has an EVENT synchronisation object associated with it. The kernel catches interrupts from the underlying hardware DMA channel (DMA_INTx) and arranges for the event to be signalled. This function is used to clear the event before waiting for an interrupt by calling SC6xDMAChannel_AwaitInterrupt. You must be careful to do this before starting the DMA

transfer or setting the DMA interrupt conditions, as setting them might trigger an unexpected interrupt.

`channel` must be a pointer returned by one of the "claim" functions listed above.

SC6xDMAChannel_AwaitInterrupt [Stand-alone]

```
#include <dma.h>
void SC6xDMAChannel_AwaitInterrupt(SC6xDMAChannel *channel);
```

Each **`SC6xDMAChannel`** has an `EVENT` synchronisation object associated with it. The kernel catches interrupts from the underlying hardware DMA channel (DMA_INTx) and arranges for the event to be signalled. This function suspends the calling thread until that event is signalled. You should clear the event with `SC6xDMAChannel_ResetEvent` before setting up the transfer and waiting for the interrupt.

`channel` must be a pointer returned by one of the "claim" functions listed above.

The following gives an outline example of using interrupts:

```
#include <dma.h>
#include <dma.h>

SC6xDMAChannel *channel;  // DMA channel to be claimed
DMA_REG *dma;             // channel's hardware registers

...claim DMA channel using claim functions ...

SC6xDMAChannel_ResetEvent(channel);    // clear channel event

dma->counter     = count;
dma->dst_address = to;
dma->src_address = from;
dma->sec_control = DMA_S_BLOCK_IE(1);  // enable block interrupt

// Start the DMA, interrupt on count expired
dma->pri_control = DMA_P_TCINT(1) | DMA_P_START(1) | ...;

...do some work while the DMA operation executes...

// Wait for the DMA to finish
SC6xDMAChannel_AwaitInterrupt(channel); // waits for DMA_INTx
```

SC6xDMAChannel_Operation [Stand-alone]

```
#include <dma.h>
void SC6xDMAChannel_Operation(SC6xDMAChannel *channel,
                              unsigned int prictrl);
```

This function is provided for the common situation where a thread has nothing to do between initiating a DMA operation and it being completed. It encapsulates the sequence:

- `SC6xDMAChannel_ResetEvent(channel);`
- Assign **`prictrl`** to the DMA primary control register for the selected channel. This value should usually have at least the **`START`** and **`TCINT`** bits set. If **`START`** is not set, the DMA operation will not start (it could be started separately later). If **`TCINT`** is not set, the DMA channel will not interrupt at the end of the transfer, and the function will therefore never return.
- `SC6xDMAChannel_AwaitInterrupt(channel);`

`channel` must be a pointer returned by one of the "claim" functions listed above.

Chapter 14. EDMA

This section assumes you are familiar with the operation of the C6000 EDMA channels, in particular, the way in which EDMA transfers can be synchronised. The Diamond kernel manages the available EDMA channels and dynamically allocates them to concurrently active inter-processor `<chan.h>` and `<link.h>` calls. User code that wants to make direct use of the channels must claim them from the kernel, complete the DMA operation, and return the channels to the kernel. Holding on to EDMA channels can seriously affect the performance of other transfers, in particular, link operations.

The following code fragment illustrates using EDMA to copy Frames blocks of 8 32-bit words from a device FIFO to memory at Buffer. The code assumes that the device asserts EXT_INT4 when it has 8 words available. The code does no error checking in order to keep it simple.

```
#include <edma.h> ❶
#include <ext_int.h>

struct EDmaControl *C = EDMA_CTRL;  // EDMA control registers
SC6xEDMAChannel    *channel;
EDMA_REG           *dma;
SC6xEDMA           *EdmaI;
SC6xExt_Int        *Ext_IntI;

EdmaI    = SC6xKernel_LocateInterface(_kernel, SIID_SC6xEDMA); ❷
Ext_IntI = SC6xKernel_LocateInterface(_kernel, SIID_SC6xExt_Int); ❷

SC6xExt_Int_Claim(Ext_IntI, 4);     // claim EXT_INT 4 ❸

dma = SC6xEDMA_Claim(EdmaI, 4, &channel); ❹

SC6xEDMA_FlushCache(EdmaI, 1, Frames*8*sizeof(int), Buffer);

dma->opt = EDMA_2DS(1) | EDMA_LINK(1) | EDMA_TCINT(1)
                       | EDMA_PRI(1)  | EDMA_SUM(1)
                       | EDMA_DUM(1)  | EDMA_TCC(4);
dma->cnt = ((Frames-1)<<16) + 8;
dma->dst = Buffer;                  // destination buffer
dma->src = DEVICE_FIFO;             // device data address
dma->idx = 0;
dma->rld = EDMA_LINK_OFFSET(EDMA_NULL_PARAM);

C->ECR   = 1<<4;                    // clear any pending events
EnableMyDevice();                   // for synchronisation
SC6xEDMAChannel_StartWait(channel); // do the transfer❺
DisableMyDevice();
SC6xEDMAChannel_Release(channel); ❻
SC6xExt_Int_Release(Ext_IntI, 4);
```

There are several things to notice about this code:

❶ `<edma.h>` declares the kernel functions used in the rest of the code and creates a reference, `_kernel`, to kernel data structures. It also contains a typedef for a structure type, **`EDMA_REG`**, which can be used to access the EDMA transfer parameters, plus macros for accessing the various fields within the EDMA registers. **`EDMA_CTRL`** is also defined to be a pointer to the hardware's block of EDMA control registers.

❷ Production code would not call `SC6xKernel_LocateInterface` for every transfer but would initialise the interface pointers once on program startup.

❸ The code is using an external interrupt line (EXT_INT4), so it needs to claim that interrupt line from the kernel to prevent it being used elsewhere. You do not need to claim an external interrupt line if you are using devices that have dedicated interrupt lines, for example, the McBSP devices which use interrupts 12..15).

❹ `SC6xEDMA_Claim` actually claims the channel (DMA4). It returns a pointer to the corresponding EDMA transfer parameters, or NULL if the requested DMA engine cannot be allocated (because it is already claimed by another thread or by the kernel for an inter-processor link communication). Using this pointer,

the example code then fills in the required values. If `SC6xEDMA_Claim` succeeds, it returns an **`SC6xEDMAChannel`** pointer via its final argument. This pointer refers to a software structure in the kernel that describes the allocated DMA channel.

❺ `SC6xEDMAChannel_StartWait` is one of the functions that can be applied to such an EDMA channel pointer. It sets up the various EDMA control registers needed to control the transfer and then suspends the calling thread until the EDMA channel interrupts at the end of the block. While the thread is suspended and the EDMA operation is executing, other threads can continue to execute on the CPU. The kernel will catch the EDMA completion interrupt, resume the suspended thread and return control to the caller.

❻ `SC6xEDMAChannel_Release` informs the kernel that a previously claimed EDMA channel is no longer required and can be returned to the kernel's pool of free channels.

There is no obligation to use `SC6xEDMAChannel_StartWait`—it is provided to make handling EDMA interrupts easier—but you are free to wait for EDMA completion either by polling (not recommended) or waiting for the interrupt yourself with (`SC6xEDMAChannel_AwaitInterrupt`). The only mandatory step is to claim the EDMA channel before attempting to touch the corresponding hardware. Failure to do so will result in mysterious hangs when your code clashes with a concurrent inter-processor `<chan.h>` or `<link.h>` call in some other task or thread, and the kernel then decides to service that call using the same EDMA channel that you are already using for something else.

EDMA Channel Availability

Different C6x processors provide different numbers of EDMA channels: the C64 has 64 while other processors have 16. As it is highly unlikely that many applications will require large numbers of EDMA channels, Diamond usually arranges for the first 16 to be made available. This minimises the amount of memory needed to support EDMA and has proved to be adequate for the kernel and the most users. However, if you do need more than 16 channels, you can request 32, 48, or the full 64. You do this by defining a new processor type and using the "`MAP=`" qualifier to identify the appropriate EDMA handler module. For example, to create a variant of an existing processor type "MyProc" with 64 EDMA channels you could define a new processor type as follows:

```
PROCESSORTYPE MyProc64 MyProc MAP=DMA:EDMA64
```

The available EDMA modules are:

EDMA16	(default) 16 channels (0..15)
EDMA32	32 channels (0..31)
EDMA48	48 channels (0..47)
EDMA64	(default) 16 channels (0..15)

EDMA events used by Diamond

Diamond uses only EDMA transfer complete codes 4..8.

Event	**Used**	**Function**
0		Host-to-DSP interrupt
1		Timer 0 interrupt
2		Timer 1 interrupt
3		EMIF SDRAM timer interrupt
4	yes	External interrupt 4
5	yes	External interrupt 5
6	yes	External interrupt 6
7	yes	External interrupt 7
8	yes	GPIO event 0 (EDMA completion)
9		GPIO event 1

Event	Used	Function
10		GPIO event 2
11		GPIO event 3
12		McBSP0 transmit event
13		McBSP0 receive event
14		McBSP1 transmit event
15		McBSP1 receive event

SC6xEDMA Functions

The first functions in this group allow you to claim EDMA channels from the kernel's pool for your own use. You must use one of these functions before touching the EDMA hardware. Each of the functions returns an (EDMA_REG *) pointer to the allocated EDMA channel's hardware registers, or NULL if the requested EDMA channel cannot be allocated. See EDMA.H in the Diamond installation folder for the names and types of the various structures' members.

SC6xEDMA_Claim [Stand-alone]

```
#include <edma.h>
EDMA_REG *SC6xEDMA_Claim(SC6xEDMA        *EdmaI,
                         int             n,
                         SC6xEDMAChannel **c);
```

This function attempts to allocate EDMA channel number **n** from the kernel's pool of free channels. If it succeeds, it returns a pointer to the requested channel's hardware control registers. If it fails, it returns NULL.

EdmaI must be a pointer to the kernel's EDMA manager interface, as returned by a call to SC6xKernel_LocateInterface with a second argument of **SIID_SC6xEDMA**.

n is the requested EDMA channel number. It must be in the range 0–15 by default. This range can be extended by selecting more EDMA channels for the processor in the configuration file.

c is a pointer to an (**SC6xEDMAChannel ***) variable. The function sets this variable to point to an EDMA channel descriptor in the kernel. This pointer is used to refer to the allocated EDMA channel when calling the functions described later.

SC6xEDMA_ClaimWait [Stand-alone]

```
#include <edma.h>
EDMA_REG *SC6xEDMA_ClaimWait(SC6xEDMA       *EdmaI,
                             int             n,
                             SC6xEDMAChannel **c);
```

This function attempts to allocate EDMA channel number n from the kernel's pool of free channels. If the requested channel is unavailable, the calling thread will be suspended until it does become available, if ever. Therefore this function will never fail simply because the requested channel is not available. When it succeeds, it returns a pointer to the requested channel's hardware control registers. If it fails for some other reason, it returns NULL. The arguments are the same as for SC6xEDMA_Claim above.

SC6xEDMA_ClaimAny [Stand-alone]

```
#include <edma.h>
EDMA_REG *SC6xEDMA_ClaimAny(SC6xEDMA         *EdmaI,
```

```
                                int              oper,
                                SC6xEDMAChannel **c);
```

This function is similar to `SC6xEDMA_Claim`, except that instead of requesting a particular channel number, the call returns a pointer to any EDMA channel that happens to be free. If no channel is presently free, or if the call fails for some other reason, it returns NULL.

`EdmaI` and **`c`** have the same meanings as for `SC6xEDMA_Claim`.

`oper` is ignored. It is retained for compatibility with the older DMA functions.

SC6xEDMA_ClaimAnyWait [Stand-alone]

```
#include <edma.h>
EDMA_REG *SC6xEDMA_ClaimAnyWait(SC6xEDMA         *EdmaI,
                                int               oper,
                                SC6xEDMAChannel **c);
```

This function is similar to `SC6xEDMA_ClaimAny`, except that if no channel is presently free, the calling thread will be suspended (indefinitely) until a channel does become free. Therefore this function will never fail simply because a channel is not available. When it succeeds, it returns a pointer to the allocated channel's hardware control registers. If it fails for some other reason, it returns NULL. The arguments are the same as for `SC6xEDMA_ClaimAny` above.

SC6xEDMA_FlushCache [Stand-alone]

```
#include <edma.h>
void SC6xEDMA_FlushCache(SC6xEDMA      *EdmaI,
                         UINT32         Modify,
                         UINT32         Bytes,
                               volatile void *Memory);
```

This function is provided to control the cache when using EDMA.

`EdmaI` must be a pointer to the kernel's EDMA manager interface, as returned by a call to `SC6xKernel_LocateInterface` with a second argument of **`SIID_SC6xEDMA`**.

`Bytes` and **`Memory`** specify the size and location of the memory to be used in a subsequent EDMA transfer.

`Modify` should be set non-zero if the EDMA transfer that is to follow will modify the memory; otherwise it should be 0.

The function will flush any data in the cache corresponding to the external memory area defined by **`Bytes`** and **`Memory`**. If **`Modify`** is non-zero, that cache data will also be invalidated. The function does nothing if internal memory is specified.

The C6000 cache on processors with EDMA is unable to maintain coherence with external memory when both the CPU and an EDMA channel access the memory. This means, for example, that it is possible for a CPU read of external memory to be satisfied with erroneous data from the cache even though an EDMA operation has written new values directly to that memory. To work round this hardware limitation it is necessary to manipulate the cache explicitly each time you perform an EDMA transfer involving external memory.

The following guidelines should be followed for each EDMA transfer:

1. Ensure that the CPU cannot access any external memory that will be associated with the EDMA transfer you are about to start. "Associated" here includes memory that falls into the same cache

line (128 bytes) as the memory specified in the EDMA transfer parameters. For example, the memory associated with a transfer of 32 bytes from $A0000030_{16}$ to $B0000120_{16}$ extends from $A0000000_{16}$ to $A000007F_{16}$ and from $B0000100_{16}$ to $B000017F_{16}$. The simplest way to do this is to align buffers on 128-byte boundaries and make them multiples of 128 bytes. This is not always possible.

2. Call `SC6xEDMA_FlushCache` before starting the transfer. You will need to call this function twice if the transfer is from external memory to external memory.

3. Start the transfer.

4. Once the transfer has completed, the CPU may safely access the affected memory.

Warning

This function can be quite slow as it must poll to detect completion of the cache operation to complete.

SC6xEDMA_ClaimParam [Stand-alone]

```
#include <edma.h>
EDMA_REG *SC6xEDMA_ClaimParam(SC6xEDMA *EdmaI,
                              UINT32    Which);
```

The parameters for EDMA operations are held in blocks of registers. This function returns a pointer to one such block. The parameter **`Which`** determines selects the block you want. If **`Which`** is 0, the function returns a pointer to the first free block it finds, otherwise the value (in the range 16 <= **`Which`** <= 84) is used to select a particular block. Parameter block 85 is reserved for use as a terminating null block (see `EDMA_NULL_PARAM`). The function returns a NULL pointer if a suitable parameter block cannot be claimed.

SC6xEDMA_ReleaseParam [Stand-alone]

```
#include <edma.h>
void SC6xEDMA_ReleaseParam(SC6xEDMA *EdmaI,
                           EDMA_REG *Param);
```

Release a parameter block for reuse. The block, Param, must have been claimed previously using `SC6xEDMA_ClaimParam`.

EDMA_LINK_OFFSET [Stand-alone]

```
#include <edma.h>
UINT32 EDMA_LINK_OFFSET(EDMA_REG *P);
```

This macro converts a pointer to a parameter block into an offset from the start of the parameter area. This offset is required in the **`link`** field of one parameter block to chain it to the next.

EDMA_EVENT_PARAM [Stand-alone]

```
#include <edma.h>
EDMA_REG *EDAM_EVENT_PARAM(int n);
```

This macro converts a parameter number (in the range 0 <= **n** <= 85) into a pointer to the corresponding parameter block.

EDMA_NULL_PARAM [Stand-alone]

```
#include <edma.h>
EDMA_REG *EDMA_NULL_PARAM;
```

This macro returns a pointer to the reserved parameter block that has been initialised to zeros. It should be used to terminate a chain of transfer requests.

SC6xEDMAChannel Functions

These functions all operate on one of the **SC6xEDMAChannel** pointers returned by the "claim" functions described above. Note that functions dealing with external devices do not set the various device enables that are necessary to allow EDMA synchronisation or CPU interrupts. Refer to your C6000 module's hardware documentation for a description of enabling events and interrupts for particular devices. EDMA termination interrupts are automatically managed for you.

SC6xEDMAChannel_Release [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_Release(SC6xEDMAChannel *channel);
```

channel must be a pointer returned by one of the claim functions listed above. The EDMA channel described by this pointer is released to the free pool.

SC6xEDMAChannel_ResetEvent [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_ResetEvent(SC6xEDMAChannel *channel);
```

Each **SC6xEDMAChannel** has an EVENT synchronisation object associated with it. The kernel catches interrupts from the underlying hardware EDMA channel (EDMA_INT) and arranges for the appropriate event to be signalled. This function is used to clear the event before waiting for an interrupt by calling SC6xEDMAChannel_AwaitInterrupt. You must be careful to do this before starting the EDMA transfer or setting the EDMA interrupt conditions, as setting them might trigger an unexpected interrupt.

channel must be a pointer returned by one of the "claim" functions listed above.

SC6xEDMAChannel_AwaitInterrupt [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_AwaitInterrupt(SC6xEDMAChannel *channel);
```

Each **SC6xDMAChannel** has an EVENT synchronisation object associated with it. The kernel catches interrupts from the underlying hardware EDMA channel (EDMA_INT) and arranges for the appropriate event to be signalled. This function suspends the calling thread until that event is signalled. You should clear the event SC6xEDMAChannel_ResetEvent before setting up the transfer and waiting for the interrupt.

channel must be a pointer returned by one of the "claim" functions listed above.

SC6xEDMAChannel_Start [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_Start(SC6xEDMAChannel *channel);
```

This function starts an EDMA transfer by setting the appropriate bit in **ESR**.

SC6xEDMAChannel_StartWait [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_StartWait(SC6xEDMAChannel *channel);
```

This function is provided for the common situation where a thread has nothing to do between initiating an EDMA operation and dealing with its completion. It encapsulates the sequence:

1. `SC6xEDMAChannel_ResetEvent`(**`channel`**);
2. Set the bits in **`CIER`** and **`EER`** corresponding to the given channel; note that **`ESR`** is not used;
3. `SC6xEDMAChannel_AwaitInterrupt`(**`channel`**);
4. Clear the bits in **`CIER`** and **`EER`** corresponding to the given **`channel`**.

`channel` must be a pointer returned by one of the "claim" functions listed above.

This function assumes that the actual transfer will be initiated by the synchronisation event associated with the EDMA channel being used. You should call `SC6xEDMAChannel_KickWait` when you want the transfer to start immediately.

SC6xEDMAChannel_KickWait [Stand-alone]

```
#include <edma.h>
void SC6xEDMAChannel_KickWait(SC6xEDMAChannel *channel);
```

This function is provided for the common case where the EDMA channel does not need to wait for a synchronisation signal before initiating a transfer. It encapsulates the sequence:

1. `SC6xEDMAChannel_ResetEvent`(**`channel`**);
2. Set the bits in **`CIER`** and **`EER`** corresponding to the given **`channel`**;
3. Set the bit in **`ESR`** corresponding to the given **`channel`** to start the transfer;
4. `SC6xEDMAChannel_AwaitInterrupt`(**`channel`**);
5. Clear the bits in **`CIER`** and **`EER`** corresponding to the given **`channel`**.

`channel` must be a pointer returned by one of the "claim" functions listed above.

Chapter 15. QDMA

Introduction

The QDMA Manager gives you access to the Quick DMA (QDMA) hardware that is present on certain C6000 processors. It presents a simple and efficient interface that permits all modes of operation of QDMA and supports different options for determining the completion of transfers.

This document assumes you have read and understood TI's documentation on EDMA and QDMA, and the chapter on the EDMA manager.

Principles of Operation

QDMA is similar to EDMA, but allows for faster transfers at the cost of restricted functionality; not all of the EDMA facilities are available with QDMA.

QDMA has a single set of five registers. These can be accessed in two ways: directly or through "pseudo registers". Writing directly simply sets a register value; writing to a pseudo register sets the register value but also initiates the transfer. On completion, the registers retain their starting values, so subsequent, similar operations can be performed by updating only the changed values, the final value being written to a pseudo register to start the transfer.

The QDMA mechanism uses EDMA techniques to indicate termination; a "transfer complete code" value is used to set a bit in the CIPR system register on completion of the operation. This bit can be polled or used to generate an interrupt. From this point of view, a QDMA transfer could be confused with an EDMA transfer. To avoid this confusion when using Diamond, claiming the QDMA channel with a specific transfer completion code will also claim the EDMA channel that uses the same bit for synchronisation. It is anticipated that QDMA will always be used with a TCC corresponding to an EDMA channel that is otherwise unused by Diamond and the application (for example, EDMA channel 15).

Warning

Note that nothing can be done to protect against a user using the same TCC in a concurrent transfer with a different EDMA channel; this will inevitably lead to program failure.

The module does not take any action to deal with potential problems that may result from the C6000 DMA processor's inability to maintain coherency between external memory and the cache; this is left to the user.

Header File

The header file QDMA.H gives access to all of the functions in the QDMA module. It also includes EDMA.H.

Status

All of the functions provided in the QDMA module return a status code of the following type:

```
typedef enum {
   QDMA_OK         =  0,   // success
   QDMA_NOTCLAIMED = -1,   // QDMA not claimed
   QDMA_RANGE      = -2,   // illegal EDMA channel
   QDMA_EDMA       = -3,   // EDMA already claimed
   QDMA_HOW        = -4    // illegal arg to Claim
} QDMA_STATUS;
```

Preparing to Transfer

You must take several steps before you can use the EDMA module:

1. First, call QDMA_Interface to obtain an interface to the QDMA module for use in subsequent QDMA calls. For example:

```
#include <qdma.h>
SC6xQDMA *QdmaI = QDMA_Interface();
if (!QdmaI) error("cannot obtain QDMA interface");
```

   You only need do this once in any task; calling it repeatedly is not wrong but will waste time. The function returns NULL if QDMA cannot be found.

2. Before using QDMA, you must claim the channel:

```
QDMA_STATUS Status;
Status = QDMA_Claim(QdmaI, 15, QDMA_POLL);
```

   This reserves the single QDMA channel on a processor for the thread's exclusive use and specifies three things:

   a. The QDMA interface obtained in step 1 above.

   b. The termination event to be used. This must be in the range 0<=N<=15 and also indicates which EDMA channel is to be claimed. You should chose an event that is not used elsewhere for a different purpose. This example uses event 15 which is never used by Diamond.

   c. The action taken once a transfer has started. There are three options:

QDMA_INTERRUPT	The module will suspend the calling thread and wait for the QDMA to indicate completion of the transfer with an interrupt. Other threads will continue to execute while the thread is waiting. It will be rescheduled when the interrupt is detected. All interrupt handling is managed by the module. In particular, the CIPR and CIER system registers must not be altered by user code.
QDMA_POLL	The module will loop testing for completion of the transfer. This should only be used for transfers that will terminate quickly, as the thread will consume CPU cycles while it waits.
QDMA_NOWAIT	Control will be passed back to the calling thread immediately; the transfer will continue to progress. It is the caller's responsibility to determine when the transfer has completed. Most users will find the interrupt or polling options more convenient

   The function can return four values:

QDMA_OK	The claim was successful.
QDMA_RANGE	The termination event is not in the range 0 <= N <= 15.
QDMA_HOW	The termination action is unknown.
QDMA_EDMAINUSE	The corresponding EDMA channel is in use.

If another thread on the processor has claimed the QDMA channel, this call will suspend the calling thread until the QDMA channel is released.

A QDMA_Claim/QDMA_Release sequence has been timed on an 150MHz SMT374_6711 as takng 5.2us.

3. Now you can perform as many QDMA transfers as you wish (see Transfers).

   If you use QDMA_NOWAIT you can explicitly test for completion using QDMA_Complete, which returns a non-zero value if the previous QDMA transfer has completed. It returns 0 otherwise. For example,

   ```
   while (QDMA_Complete(QdmaI)==0) {}
   ```

4. Finally, you can release all resources held by the QDMA channel (including the associated EDMA channel) by calling QDMA_Release. This makes QDMA available for use by other threads. The interface remains valid for further use.

   For example:

   ```
   QDMA_Release(I);
   ```

   The function can return two values:

QDMA_OK	The release was successful.
QDMA_UNCLAIMED	The QDMA channel had not been claimed.

If only one thread on a processor needs to use QDMA, it may claim the channel once and never release it.

If you wish to change the termination action (from QDMA_POLL to QDMA_INTERRUPT, for example) you must release the channel and then claim it again.

Transfers

QDMA transfers are based around the following structure:

```
typedef struct {
   UINT32  Opt;
   void   *Src;
   UINT32  Cnt;
   void   *Dst;
   UINT32  Idx;
} QDMA_REGS;
```

Object of this type are used to hold the parameters for a QDMA operation. The details of the fields and their use are fully described in the TI QDMA and EDMA documentation (SPRU190).

Transfers are usually started using the following function:

```
QDMA_STATUS QDMA_Perform(SC6xQDMA *I, QDMA_REGS *R);
```

The values in the QDMA_REGS structure, R, are assigned to the corresponding QDMA parameter registers. The Opt value is assigned last using a pseudo register to initiate the transfer. Depending on the way the QDMA channel had been claimed, this call will either wait for the transfer to complete (QDMA_INTERRUPT or

QDMA_POLL) or return immediately (QDMA_NOWAIT).

The value specified for Opt will be modified before assignment as follows:

1. The TCC (Transfer complete code) field will be set to the value of the termination event specified when the channel was claimed.
2. The TCCM field will be set to 0.
3. The TCINT (Transfer complete interrupt) bit will be set to allow CIPR to be used to indicate completion of the transfer.

This modification of Opt cannot be done if you write to the Opt register explicitly. In this case it is your responsibility to ensure that the value written has the correct values in the TCC, TCCM, and TCINT fields.

The function can return two values:

QDMA_OK	The call was successful.
QDMA_UNCLAIMED	The QDMA channel had not been claimed.

Starting a QDMA transfer while another is in progress will lead to unpredictable behaviour.

QDMA Registers

The hardware's QDMA registers may be accessed using the macro QDMA_REGISTERS that is defined by `<QDMA.H>` . Note that the QDMA registers may only be written; any values returned by reading them are undefined. The fields correspond to the QDMA registers described in the TI documentation. Explicitly changing Opt requires you to ensure that the value has the TCC, TCCM and TCINT fields set as for QDMA_Perform. This will be done for you if you set Opt using QDMA_Restart (see below).

A transfer can be started by assigning the final or only changed value to a pseudo register using the macro QDMA_Restart. You should not touch the QDMA pseudo registers explicitly; these are managed for you by Diamond.

The macro takes three parameters: QDMA_Restart(I, R, V):

I	The QDMA Interface
R	The register to be modified
V	The value to assign to the register

For example, assuming a previous transfer had been carried out using QDMA_Perform:

```
QDMA_REGS *Q = QDMA_REGISTERS;
Q->Src = &input_buffer;
QDMA_Restart(QdmaI, Q->Dst, &output_buffer);
```

This starts the transfer by assigning Q->Dst and then waits for completion in the same way as QDMA_Perform. It uses the same return values.

A QDMA Example

```
// This example is intended to give a flavour of how to
// use QDMA. It does not attempt to do any error
// detection.
```

```
// A complete example is in the installation package.

#include <QDMA.h>

#define SIZE 1024

typedef struct {
   float real;
   float imag;
} COMPLEX;

// Build a complex structure from two array of float
// The values in the arrays Real and Imag are
// interlaced into the array C.
void Combine(float *Real, float *Imag, COMPLEX *C)
{
   SC6xQDMA *QdmaI = QDMA_Interface();
   QDMA_REGS R;
   QDMA_REGS *Q = QDMA_REGISTERS;
   QDMA_Claim(QdmaI, 15, QDMA_INTERRUPT);  // use TCC 15

   // move in the real parts
   R.Src = Real;
   R.Dst = &C->real;
   R.Cnt = (SIZE<<16) | 4;          // size * 4 bytes
   R.Idx = 8;                       // every other word
   R.Opt = EDMA_SUM(1)
         | EDMA_DUM(3)
         | EDMA_PRI(1);
   QDMA_Perform(QdmaI, &R);

   // move in the imaginary parts
   Q->Src = Imag;
   QDMA_Restart(QDmaI, Q->Dst, &C->Imag);

   QDMA_Release(QdmaI);             // done
}
```

Chapter 16. Troubleshooting

This chapter is designed to help those who are debugging Diamond applications, and is designed to be used in conjunction with the Index at the end of this User Guide. It lists a number of symptoms, together with likely causes and possible remedial actions. Each of the listed symptoms may be found under the symptoms of problems entry in the Index; the heading of each possible cause may found under causes of problems. By looking up these entries, you can find references to pages of the manual where more details may be found.

In addition, the following sections of the manual deal with the error messages generated by some of the utilities.

- The configurer, config.
- The server, WS3L.

My application does not run

This is an initial checklist to follow if you are unable to get the server to run your application.

- You should start by running any utilities provided by your DSP vendor to check that the hardware and associated device drivers have been installed correctly and are functioning.
- Can you run the hello word example from the Diamond examples folder? If you can, the problem lies in your application. If you can't, there is likely to be some problem with your DSP hardware. Check the following points for further suggestions.
- Have you changed the server's Standard I/O options? Redirecting all output to a file rather than the screen can make an application appear not to run, for example. If in doubt, reset the options to the safe default state by selecting View/Options and pressing Reset to Default Options.
- Does your hardware appear to be working? Many DSP boards have indicators (LEDs) that display some processor state. Check with your board documentation that these are showing the expected state.
- Is the board's power supply correct? Some DSP modules need to be screwed down to the carrier board to receive correct power.
- Does your application load? You can check this from the server by selecting View/Options/Monitoring/General Monitoring and trying to run your application. You should see three monitoring messages:

  ```
  MON: Resetting DSP modules
  MON: Loading your application
  MON: Loading completed
  ```

  If these messages do not all appear, check the following:

  - Has your application been built properly?
  - Have you used the Diamond command 3L? Tasks built using the recipes in TI's manuals will not work under Diamond.
  - Did you build your application for the correct processors? Applications compiled for the 62xx processors (3L c, 3L t) will work on both C62xx and C67xx processors. Applications built for c67xx processors (3L c67, 3L t67) will not execute on c62xx processors.
  - Were there any error messages from the configurer when you built the .app file? You should correct any problems here first.
  - Do the processor types in your configuration file match the processors in your DSP system?
  - Is your DSP network connected properly?

- Does the number of processors in your hardware system correspond to the number of PROCESSOR statements in the configuration file used to build the application?
- Are the processors connected in the way described by the WIRE statements in your configuration file? External cables, pre-defined links, or programmable links can connect processors. Check that all necessary connections have been made correctly.
- Some PC boards have a distinction between a "root" board, which has a link to the PC, and "non-root" boards, which have no such connection. Check that any necessary switches on the boards are set in the appropriate ways. Also, note that some boards give you access to the link used to communicate with the host. Make sure that you have not connected this link on the root processor's board to any other link.
- Can the server see all the DSP boards you expect? Check using Board/Select.
- If you have more than one DSP board in your system, have you selected the correct one? The server shows the selected board at the bottom right of its window.

- Have you given enough memory to your tasks? Check the TASK statements in your configuration file.
- Have you used the appropriate #include files in your source programs? A common mistake is to forget to include `<stdio.h>` . Also, check that you are getting the Diamond include files and not TI's.
- Is some other application using the same DSP system? This could be another instance of the server or a board maintenance application from your board vendor. Under certain rare circumstances, it is possible to kill the server's user interface without shutting down the server properly. Stop the server and check using the Task Manager (Processes) that there aren't any instances of WS3L.exe running. You can safely stop any you find (End Process).
- Have you been using Code Composer and left any of the DSP processors halted? If you have been using Code Composer, start it again, and for each processor: halt it if necessary, and then select "Run Free". Try running your application again.
- Check that your application is starting to run by putting a printf at the start of main in one of the tasks on the root processor. If this generates output, the problem lies further on in your code. Some common errors are:
  - Using semaphores before they have been initialised with sema_init.
  - Using events before they have been initialised with event_reset.
  - Using a local channel before it has been initialised with chan_init. Note that you must not initialise the channels passed in as parameters to main.
  - Forgetting to use the appropriate header files, notably `<stdio.h>` , and `<thread.h>` .

Compilation, Linking, Configuration

compiler cannot be found

search path not set correctly

The PATH environment variable must include the folder where Diamond has been installed.

compiler cannot find header files

compiler invoked without Diamond headers

Applications must be compiled using the header files supplied with Diamond. The most likely cause for this is not using the Diamond command (3L) but using the compiler directly (cl6x) and not specifying that the Diamond installation folder be searched first (-I"C:\3L\Diamond\bin\c6000\Sundance\include").

relocation errors

configurer produces relocation errors

The TI compiler can generate references to external objects in two ways: using one instruction or using two. References using single instructions are faster than those using two instructions but they have limited addressing capabilities: objects that are too far away or are too large may not be reachable. References using two instructions can reach anywhere in the processor's address space. "Relocation errors" indicate that the configurer has discovered a single-instruction reference that cannot reach its target. To ensure the compiler generates the longer instruction sequences when necessary, you should ensure that your source declares the objects as "far". For example

```
extern far int Counter;
                  extern far void SpecialProcessing(void);
```

This usually only happens when you explicitly put data or functions in separate sections and not in the usual ".data" or ".text" sections.

linker complains about relocations

The TI compiler and linker must agree on the format and content of object files. On a number of occasions, TI has updated the code generation tools in a way that leaves them incompatible with older versions. Usually the problem is limited to older versions not accepting output generated by the newer versions. This problem can appear if you are using a version of the Diamond libraries built with a version of the tools that is newer than the tools you have installed. There are only two solutions:

1. upgrade to the new version of the TI tools;
2. revert to an older version of Diamond

wrong version of software executed

search path not set correctly

Check that your PATH variable includes the Diamond installation folder, and that it comes before any other folder containing any program with the same name as a Diamond command.

Complete Failure at Run Time

application hangs or runs wild

wrong processor type given in configuration file

When you run an application, the hardware used must match the trouble in the configuration file. In particular, the TYPE information on the PROCESSOR statements must correctly identify each processor. If you have used the default processor type, check using the ProcType command that would have selected the correct processor type.

channel message has incompatible lengths

When you are sending a message through a channel, whether or not this is routed through a link to

another processor, the sending and receiving sides must agree on the length of the message.

channel transfer on uninitialised channel

Before an internal channel is used, it must be initialised using the chan_init function.

wrong header files used

Diamond has its own set of header files. These are, in general, incompatible with those supplied with the Texas Instruments C compiler. To make sure the compiler picks up the correct header files, you should use the Diamond command (3L c) to compile programs.

function prototypes necessary

You must supply a prototype for any function to which you wish to make external or forward references. If you do not do this, the compiler may generate the wrong code for the function call, but even so it will not report an error. This is particularly serious if the function has a variable number of parameters (printf, for example). You should always include header files for library functions.

ill-advised alterations to CPU registers

You should only change any of the CPU registers if you are certain about what you are doing. It is possible to disrupt the microkernel in this way.

multiple use of run-time library

Several parts of the run-time library cannot be used by more than one thread from the same task at a time. In a multi-threaded task, you should wait for the semaphore par_sema before using the run-time library, or use the special protected versions of certain functions, which are defined in `<par.h>` . You do not need to protect calls from different tasks in this way.

multiple use of shared object

If two or more threads are using the same data area, link, etc, it should be protected by a semaphore or by some other technique, to ensure that anomalies do not occur.

no memory assigned to STACK or HEAP

If you assign a size to either the HEAP or STACK logical area in a TASK statement in your configuration, but do not assign a size to the other, the other will in fact be given no memory. Nearly all tasks will fail under these conditions. Either assign values to both or use the DATA logical area, which is shared by the two of them. One task on each processor may have no data assignments; this task will be given all the available remaining memory for its DATA logical area.

not enough memory assigned to logical area

If the stack or heap overflows its assigned memory area, the task will usually fail.

par_sema already claimed when thread_new used

The thread_new function gets space from the heap using par_malloc, which waits for the par_sema semaphore. If the thread that calls thread_new has already claimed par_sema, par_malloc will wait forever to claim par_sema, although the thread has already got it, and as a result thread_new will never return.

tried to turn interrupts off

Diamond applications usually run with the processor interrupts turned on. Turning interrupts off at the wrong time will often prevent your application from running.

two threads waiting on one channel

If a thread tries to do a read on a channel, the thread will wait until another thread does a write on it; at that moment, the transfer will take place. If, during the time the thread is waiting for the transfer to take place, a third thread tries to do a read on the same channel, the effects are unpredictable.

The same applies if a thread tries to do a write on a channel, and is waiting for another thread to read the data; if a third thread tries to write to the channel, the effects are unpredictable.

uninitialised semaphore

You must always initialise a semaphore, dynamically using the `sema_init` function or statically using `static_sema_init`, before using any other function on it, or before attaching it to an interrupt. Note that the run-time library initialises par_sema for you.

using channel in both directions at once

Two threads should not issue reads on a channel at the same time. The same applies to writes.

failing to claim a DMA (or EDMA) channel

Diamond can claim and release DMA channels dynamically. If you use a channel without first claiming it, another part of your application may be given that same channel to use, leading to undefined behaviour. Diamond itself will only ever use DMA channels corresponding to the four external interrupt lines: INT4..INT7, but application code is free to use any.

application will not load or start

hardware configuration trouble incorrect

The hardware part of a configuration file, that is, the PROCESSOR and WIRE statements, must be an accurate trouble of the network on which the application is going to run. In particular, the TYPE of each processor in the configuration file must match the actual type of processor being used.

task placed on processor of wrong type

If you are building a mixed-processor application, it is your responsibility to ensure that tasks are

placed on processors of the correct type. Obviously, a task placed on the wrong type of processor will not run at all.

communication with host disrupted

connection to host closed

When the main function of a task linked against the full run-time library returns, the connection to the host server is closed. Any threads created by that task will no longer be able to perform any host I/O, for example, printf. If you do have active I/O threads, the main function should not return but terminate by calling thread_stop.

misuse of link.h functions

The link communication functions in <link.h> should normally be used only on links mentioned in DUMMY WIRE statements. Using these statements on the link to the host PC will almost certainly interfere with the transmission protocol.

multiple use of run-time library

Many parts of the run-time library cannot be used by more than one thread at a time. In a multi-threaded task, you should wait for the semaphore par_sema before using the run-time library, or use the special protected versions of certain functions, which are defined in <par.h> . This is described here.

processor locks up

channel message has incompatible lengths

When you are sending a message through a channel, whether or not this is routed through a link to another processor, the sending and receiving sides must agree on the length of the message.

channel transfer on badly bound port

It is possible to pass a parameter to a task by binding one of its ports to an integer value, using the configuration language BIND command. The task should not then try to do a channel transfer through this port.

ill-advised alterations to CPU registers

You should only change any of the CPU registers if you are certain about what you are doing. It is possible to disrupt the microkernel in this way.

multiple use of run-time library

Many parts of the run-time library cannot be used by more than one thread at a time. In a multi-threaded task, you should wait for the semaphore par_sema before using the run-time library; or use the special protected versions of certain functions, which are defined in <par.h> .

multiple use of shared object

If two or more threads are using the same data area, link, etc, it should be protected by a semaphore or by some other technique, to ensure that anomalies do not occur.

no memory assigned to STACK or HEAP

If you assign a size to either the HEAP or STACK logical area in a TASK statement in your configuration, but do not assign a size to the other, the other will in fact be given no memory. Nearly all tasks will fail under these conditions. Either assign values to both or use the DATA logical area, which is shared by the two of them. One task on each processor may have no data assignments; this task will be given all the available remaining memory for its DATA logical area.

not enough memory assigned to logical area

If the stack or heap overflows the assigned memory area, the task will usually fail.

two threads waiting on one channel

If a thread tries to do a read on a channel, the thread will wait until another thread does a write on it; at that moment, the transfer will take place. If, during the time the thread is waiting for the transfer to take place, a third thread tries to do a read on the same channel, the effects are unpredictable.

The same applies if a thread tries to do a write on a channel, and has to wait for another thread to read the data; if a third thread tries to write to the channel, the effects are unpredictable.

using channel in both directions at once

Two threads should not issue reads on a channel at the same time. The same applies to writes.

server hangs or runs wild

host communication disrupted

Check that you have not used the `<link.h>` functions on links that have been mentioned in the configuration file.

multiple threads accessing the server

Check that no two functions marked Server in the list of functions can be executing in different threads at the same time. See `<par.h>` .

system started another Diamond application

The system function should not be used to start another Diamond application. Doing so will overwrite the one that is running and disturb the server.

ANSI Functions

data in file seem to be corrupt

accessing binary file via redirection

The only kind of file which may be specified with the command line "<" and ">" redirection operators is a text file.

EDOM set in errno

arguments of strtol or strtoul are wrong

See strtol (and following) for details of the arguments required.

mathematical function argument out of range

Several mathematical functions do this

end of file corrupt or absent

application ends while I/O unfinished

When a task's main function exits, either by returning or calling exit, the task is stopped. If the task uses C standard I/O, the run-time library winds down the I/O system and closes any open files. If there are other threads that have not finished their I/O, the files they are using may be incomplete.

multiple use of run-time library

Many parts of the run-time library cannot be used by more than one thread at a time. In a multi-threaded task, you should wait for the semaphore par_sema before using the run-time library; or use the special protected versions of certain functions, which are defined in `<par.h>` .

task unilaterally stops application

In an multi-task application, any task can terminate the whole application by calling _server_terminate_now. If other tasks have not completed their I/O, data may be lost and files corrupted.

ERANGE set in errno

ASCII number out of range for double

If the string argument of the strtod function represents a number that is outside the range for a double, ERANGE is set.

mathematical function result out of range

Several mathematical functions do this

file position is wrong

file position functions cannot be used with text files

In the current version of Diamond, the fgetpos, fsetpos, ftell and fseek functions do not work on text files, with certain exceptions that are detailed in the troubles of the functions. This is because the operating system represents newline in a file by two characters (carriage return and line feed) while C interprets this pair as a single "\n" character.

I/O behaves unexpectedly

switching file directly from input to output

If you change directly from input to output on the same file, or vice versa, you are likely read or write erroneous data. You must insert a call to fseek between the two.

I/O function returns negative value

printf and other printing functions return a negative value as an indication that an error has occurred.

I/O function returns non-zero value

Certain functions, including fclose, ferror, fgetpos, remove and rename, indicate an error by returning a non-zero value. See the documentation of the various functions for details.

I/O function returns zero

The fread and fwrite functions return zero as an error indicator (fread uses zero to indicate end-of-file as well).

I/O function returns EOF

A number of functions, including fgetc, fflush, fputc, fscanf, getc, getchar, putchar, putc, and scanf, return EOF as an error indicator. The input functions among them also return EOF to indicate end-of-file. In addition:

ungetc cannot un-get a character

See the documentation of ungetc.

I/O function returns NULL

The functions gets, fgets, fopen, and freopen return NULL to indicate an error. The same value is returned by gets and fgets to indicate an end-of-file.

NULL returned when allocating memory

heap has run out of memory

The functions calloc, malloc, realloc and memalign return NULL if they cannot allocate the space which was requested from the heap. You may need to consider expanding the amount of space assigned to the HEAP logical area in your TASK statement.

output does not appear or is corrupt

application ends while I/O unfinished

When a task's main function exits, the task is stopped. If the task uses C standard I/O, the run-time library winds down the I/O system and closes any open files. If there are other threads that have not finished their I/O, the files they are using may be incomplete.

multiple use of run-time library

Many parts of the run-time library cannot be used by more than one thread at a time. In a multi-threaded task, you should wait for the semaphore par_sema before using the run-time library; or use the special protected versions of certain functions, which are defined in `<par.h>` .

task unilaterally stops application

In a multi-task application, any task can terminate the whole application by calling _server_terminate_now. If other tasks have not completed their I/O, data may be lost and files corrupted.

time function returns wrong time

TZ environment variable not defined

Under DOS, the TZ environment must be defined to specify your time zone. See the discussion of the time function.

variable corrupt

arguments of memcpy overlap

If its arguments overlap, the behaviour of memcpy is undefined.

arguments of strcpy overlap

If its arguments overlap, the behaviour of strcpy is undefined

Parallel and Other Functions

channel transfer fails

multiple use of shared object

If two or more threads are using the same data area, link, etc, it should be protected by a semaphore or by some other technique, to ensure that anomalies do not occur. See: semaphores and volatile.

two threads waiting on one channel

If a thread tries to do a read on a channel, the thread will wait until another thread does a write on it; at that moment, the transfer will take place. If, during the time the thread is waiting for the transfer to take place, a third thread tries to do a read on the same channel, the effects are unpredictable.

using channel in both directions at once

Two threads should not issue reads on a channel at the same time. The same applies to writes.

link functions do not work

link functions do not transfer any data

Some links do not have drivers built into the kernel. In order for the appropriate driver module to be loaded, the configurer must know that one of these links will be used. It determines this by observing the link being used in WIRE statements. If you do not mention the link, the driver will not be loaded and the link functions will fail.

thread cannot see changes to shared data

shared variables may need to be volatile

The compiler may arrange for a variable's value to be held in a register. If so, changes to that variable would be invisible to another thread. If you declare the variable as volatile, the compiler will keep the variable in memory.

Note that even if you use volatile you may still need to use semaphores as well to synchronise the threads' access to a shared variable.

thread hangs

calling `<par.h>` function when par_sema already claimed

If the thread that calls one of the `<par.h>` functions has already claimed the par_sema semaphore itself, the function will never return. This is because it waits forever in vain for par_sema to be signalled.

thread_new does not return

par_sema already claimed when thread_new used

This is an example of the condition described in the previous entry. The thread_new function in fact calls par_malloc.

thread_new returns NULL

heap has run out of memory

You should consider increasing the amount of space allocated to the HEAP logical area in your TASK statement.

workspace too small

thread_new will refuse to start a thread if it is being given a workspace smaller than THREAD_MIN_STACK.

<timer.h> functions do not work

hardware clock stopped

The timer functions assume that the hardware time is running under the control of the kernel. User modification of the timer registers can disrupt timer functions and scheduling behaviour.

hardware clock disabled

The processor's clock can be disabled (stopped) by specifying CLOCK=OFF or CLOCK=0 as an attribute of the PROCESSOR statement in your configuration file.

variable corrupt

multiple use of shared object

If two or more threads are using the same data area, link, etc, it should be protected by a semaphore or by some other technique, to ensure that anomalies do not occur.

Part III. Sundance Reference

Sundance-specific Features

Table of Contents

Chapter 17. Links

Sundance TIMs come with a variety of I/O devices, some of which may be used as links.

Summary

Link	Connection
[0]	comport 0
[1]	comport 1
[2]	comport 2
[3]	comport 3
[4]	comport 4
[5]	comport 5
[6]	SDB 0
[7]	SDB 1
[8]	SDB 2 (if present)
[9]	SDB 3 (if present)

Note

Diamond will always allocate link numbers 0 to 5 for comports even though some processor TIMs do not implement all six devices, ; similarly links 6..9 are allocated to SDBs regardless of the number appearing on the TIM. Links associated with unimplemented devices should not be used as their behaviour is not defined.

Comports

Comports provide the fundamental mechanism for interconnecting Sundance modules. Every processor TIM has a number of comports that can be connected with external wires or with facilities built into carrier boards holding multiple TIMs. You should refer to the documentation for your particular types of TIM for details.

Diamond reserves links 0 to 5 for comports and these correspond directly to the six comports defined by the TIM standard.

```
PROCESSOR root SMT361
PROCESSOR node SMT361
WIRE ? root[1] node[4]   ! comports 1 and 4 are connected
```

Comports are documented to operate at a maximum speed of 20MB/s, but some implementations may be a little faster, particularly those operating between processors in a multi-processor TIM. This maximum performance is usually only achieved for transfers of 1KB or more.

SDBs

Most TIMs provide two or more Sundance Digital Busses (SDBs), which appear as links 6 to 9. SDBs must be connected using external cables.

Newer SDB links can transfer data at rates in excess of 300MB/s for blocksizes over about 4KB.

Warning

You should not attempt to mix using SDBs as links and driving them directly with the SDB functions. Doing so will certainly cause your application to fail.

Link Connection Restrictions

1. You may only connect links of the same type: the configurer will complain if you attempt to connect a comport link to an SDB link.

2. SDB links cannot participate in loading; any WIRE statements mentioning SDB link numbers will assume the attribute NOBOOT.

   You should ensure that all of your processors are connected together with comports in addition to any SDB links you may have. If you do not do this, the configurer will fail reporting an error.

Link Performance

The performance figures quoted above are maxima; the actual rates achieved in practice depend on several things:

- The size of the transfer. There is a fixed overhead associated with managing any transfer. This overhead becomes negligible when moving large amounts of data, but tends to dominate small transfers.

- Availability of DMA. When you start a transfer,the kernel will attempt to allocate a DMA channel to manage moving the data. The number of DMA channels available is limited to three. There are only four external interrupt lines and one needs to be used to synchronise each DMA transfer. One of these lines must be reserved to act as a fallback for transfers that cannot use DMA and must be managed with CPU interrupts. If all available DMA channels are tied up with concurrent transfers or have been claimed for other uses in the application, the transfer will have to use CPU interrupts and will achieve a considerably reduced transfer rate.

- The alignment of the data. Because of the inability of the C6000 DMA to maintain choerency between the cache and external memory, the Diamond kernel has to limit the use of DMA to transfers or sections of transfers that are aligned on a cache-line boundary (addresses which are a multiple of 128) and of a size that is a multiple of the cache-line size (128 bytes). Parts of a transfer that are mis-aligned or not a multiple of 128 bytes will be handled by CPU interrupts, and this is considerably slower than using DMA.

The configurer will normally choose the links to be used to carry channels created by CONNECT statements. If you need a connection to use a particular wire, to make sure it uses a fast SDB link or a slower comport link, you should explicitly PLACE the connection on the wire.

For example:

```
processor root default
processor node default

wire   ? root[1] node[4]
wire sdb root[7] node[7]   ! the SDB link

task rtask data=80K ins=1 outs=1
task ntask data=80K ins=1 outs=1

connect c1 rtask[0] ntask[0] physical
connect c2 ntask[0] rtask[0] physical

place rtask root
place ntask node
```

```
place c1 sdb                ! use the fast link
place c2 sdb
```

Connecting to Devices

Sometimes you may wish to communicate with a device (such as an ADC) or a processor that is not to be considered part of the Diamond configuration. To do this you should give a WIRE statement that references a built-in link specifier called DUMMY. This is most important when using SDB links. The WIRE statement gives the kernel the information it needs in order to set the initial direction of an SDB to receiving or transmitting. Also, the configurer uses references to SDB links as a request to load the kernel module that handles SDBs; without this module you will be unable to mention SDB links in the `link.h` functions. The DUMMY processor reference does not specify any link number.

For example:

```
PROCESSOR root SMT361
WIRE one root[6] DUMMY  ! initialise SDB link 6 as a transmitter
WIRE two DUMMY root[7]  ! initialise SDB link 7 as a receiver
```

Chapter 18. Debugging

Overview

In general, the best way to debug a multiprocessor application is to design it for debugging. While it is common practice to debug a sequential program by observing its behaviour with a debugger, that technique is, at best, tricky when two processors are involved. The asynchronous nature of multiprocessor systems makes the technique become rapidly unworkable as the number of processors increases. A more structured method is needed. One line of attack is to use tasks as a natural unit for debugging. They are independent of other tasks in the system and so it is usually possible to design dummy modules to connect to the task and provide test inputs and check outputs. It is then reasonable to use Texas Instruments' Code Composer to debug individual Diamond tasks.

Code Composer has been designed assuming that a processor can only be running a single program, but Diamond applications are different:

- They may have multiple tasks on each processor;
- Tasks may spawn multiple threads;
- Different tasks may use the same identifiers—for example, every Diamond task has a main function;
- There is a kernel executing on each processor;
- The kernel will pass control from one task to another as the application executes.

This complicates the debugging process a little, but it is still straightforward. You need to be aware of a few things first:

- At any time, one task will be the focus of debugging;
- As you debug it will be possible to see the source and identifiers for only the one task currently in focus;
- You can switch focus to another task at any time by loading its symbol table;
- If control passes from the task in focus into a different task (or into the kernel), the debugger will switch into assembly language mode;
- Breakpoints will be honoured irrespective of the current focus;
- A breakpoint set in a thread will be honoured by any thread reaching that point.

Starting to Debug

Before starting to debug a Diamond application, ensure that Code Composer is installed and that you can connect to and debug the target C6000. Some DSP boards have on-board emulators that let you debug them directly, others need external hardware, such as TI's XDS510. Note that before Code Composer is started for the first time after each power-on, you may need to run a board-specific initialisation program. Refer to the board vendor's documentation for details.

To debug a Diamond task using Code Composer:

1. Recompile all modules with debug information generation enabled. This is now done for you automatically by the **3L c** command and its variants. [1]

   You may wish to disable optimisation; to do this you will need to create a variant of the "**c**" function for the **3L** command.

2. Link each task as usual.

3. Configure the application with the -G option:

   ```
   ► 3L a test -G
   ```

[1] TI is in the process of moving over to a new format of debugging information in object files. For stability, the **3L c** command requests that the information is produced in the older COFF format. This will change in future versions of Diamond.

This instructs the configurer to emit a debug symbol table file for each task. The symbol table file will have the name that appears in the TASK statement in your configuration file with **`.out`** appended. The configurer will also generate debug files for the system components, mainly the kernel; these are usually of little interest to most users. The following example will result in the creation of a symbol table file called "first.out":

```
TASK first FILE= "second.tsk"
```

4. Start Code Composer and connect to the target board or boards. If no Diamond application has been run since the board was powered up, you may need to run a board-specific initialisation program first.
5. Go to the File menu and select Load Symbol to load the symbol table (.out) file for the first task of interest.
6. Let the processor run by selecting Run Free from the Debug menu.
7. Repeat steps 5 and 6 for every processor you wish to debug.
8. Start the server WS3L if it is not already running.
9. Select debug mode by clicking on View/Options, selecting the Parameters tab, and ticking the Debug application box.
10. Select your application and start it running. If you have more than one DSP board in your PC you may have to select the correct one before starting your application.
11. Wait for the Paused message to appear.
12. For each processor of interest:
    a. Halt the processor from the Debug menu;
    b. Select Go Main from the Debug menu;
    c. Return to the server window and hit OK in the Paused window. Debugging can now start with Code Composer.

Warning

Always select Run Free from the Debug menu before leaving Code Composer. If you forget to do this, your board may not respond. If this happens, go back into Code Composer and select Run Free for all the processors.

Notes

1. Only ever run a Diamond application when all processors are in the "run free" state. If you do not do this, Code Composer can get confused and become unable to access your board. If this happens you should shut down Code Composer and reset your board and its JTAG connection. Some manufacturers provide a program to do this (usually accessible through the server: Board/Properties), but as a last resort, power-cycling the board generally puts the JTAG back into a clean state. You may then restart Code Composer.

2. You cannot restart an application using Code Composer's restart facility; you must set the processors running free, stop the application from the Server, and restart it from step 10 above.

3. When using the Code Composer Setup utility, the processors are listed in their order on the JTAG scan chain. On some boards this is the reverse of the order in which processor numbers (or letters) are assigned in the board vendor's documentation.

4. The server does not support the debug option when booting heterogeneous systems (for instance, mixed C6000 and C4x systems). Applications for such systems must be configured with the -a option, which suppresses C4x debugging.

Chapter 19. Application Loading

A Diamond application is loaded into the root processor, and then on to any other processors in the network, in one of two ways:

1. down a link to the root processor from a host processor; or
2. from a ROM on the root processor.

Host Loading

A host processor (usually a PC) will write the whole of an application file to the host link. The bootloader on each processor (initially the root processor) will receive data on its boot link and will respond as follows:

1. The first word coming down the boot link is examined:

   - If the first word has the value $\texttt{CC000002}_{16}$, the bootloader will read in the next word which will be the type code for this processor as derived from the type given the configuration file. The bootloader will compare this code against the actual type of its processor. A two-word response is then written to the boot link:
     - If the types are not compatible, the bootloader will send a failure response by writing the value $\texttt{AA000000}_{16}$ followed by the processor's actual type code.
     - If the types are compatible, the bootloader sends a success response: the value $\texttt{AA000003}_{16}$ followed by the processor's actual type code. Both of the two input words are then discarded. The processor is marked as CHECKING.

   - If the first word has the value $\texttt{CC000004}_{16}$, the bootloader will read in the next word and discard both input words. The processor is marked as NOT_CHECKING. $\texttt{CC000004}_{16}$ is the value put in the application file by the configurer when checking is enabled; it will be changed to $\texttt{CC000002}_{16}$ by the host loader to activate checking. This is done to allow user-written host code to load applications without having to worry about load checking.

   - If the first word has neither value, the bootloader takes no specific action, and the first word is passed on to the next step in loading. The processor is marked as NOT_CHECKING.

2. Most of the subsequent words make up the information the bootloader uses to load the code and data needed by its processor. After loading this, control is passed to the Diamond kernel which then starts its own loader.

3. The Diamond loader now interprets the rest of the incoming data as a sequence of commands, each with optional data. Commands are either for this processor or are to be passed down a link to other processors.

   There are three commands of particular interest here:

   CMD_BOOT — This command simply sends its data down a selected link, just like the host sent the application file to the root. The receiving processor deals with it in exactly the same way as the root processed the data it received. This boot data will not start with the $\texttt{CC00000X}_{16}$ values described above.

   CMD_CHECK — This command is used to check that a processor about to be loaded is present and has the correct type. It will be ignored unless the processor is marked as CHECKING. It sends a pair of words (described above) down the selected link and observes the response. Loading continues if a success response is received. If an error is detected (no response within a short time or a failure response), the rest of the input from the boot link is absorbed and an appropriate failure response is written to the boot link.

   CMD_GO — This is always the final command sent to a processor. If the processor is marked as CHECKING, a success response is written to the boot link. All of the tasks on the processor are then started and the Diamond loader stops. Note that CMD_GO

commands will have been sent to all processors booted through the current one before that processor receives its own CMD_GO.

Load Checking

>The information needed to perform the checks mentioned in the previous section is added to the application file by the configurer. The checks are designed to ensure that the network of DSPs being used matches the network described in the configuration file closely enough to allow the application to be loaded. Note that only the following things are checked:

- the types of the processors mentioned in the configuration file; processors that only differ in the provision of external memory will be considered equivalent providing the actual processor has no less memory than the processor declared in the configuration file.
- the links that are used during the loading process; links that are not used during loading are not checked.

Requirement

Load checking is only supported by versions of the Sundance bootloaders dated 27 February 2004 or later. If you attempt to load an application containing checking information to a TIM with an old bootloader, the system will hang. Contact Sundance for information about updating your flash ROM with the latest bootloaders. Load checking can be inhibited by using an OPTION statement in the configuration file.

Default state

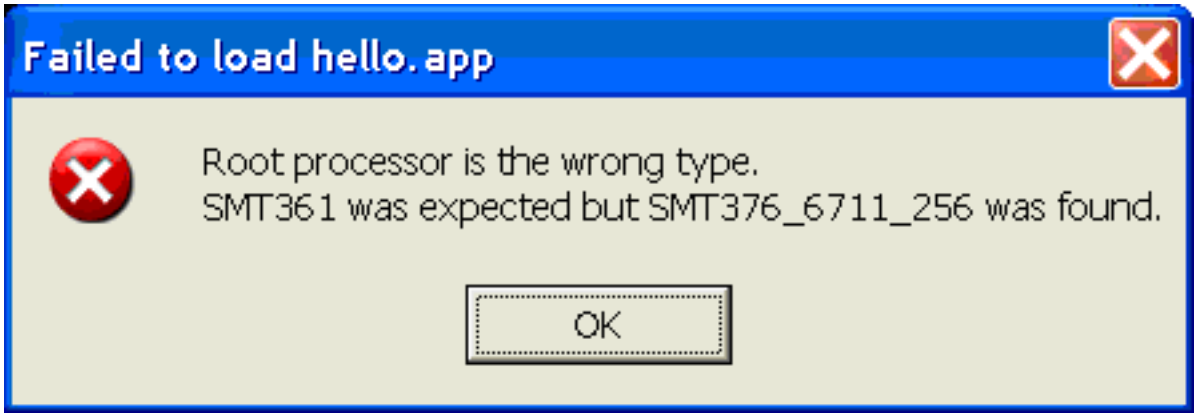
The default state is for load checking to be enabled.

Loading Checks Performed

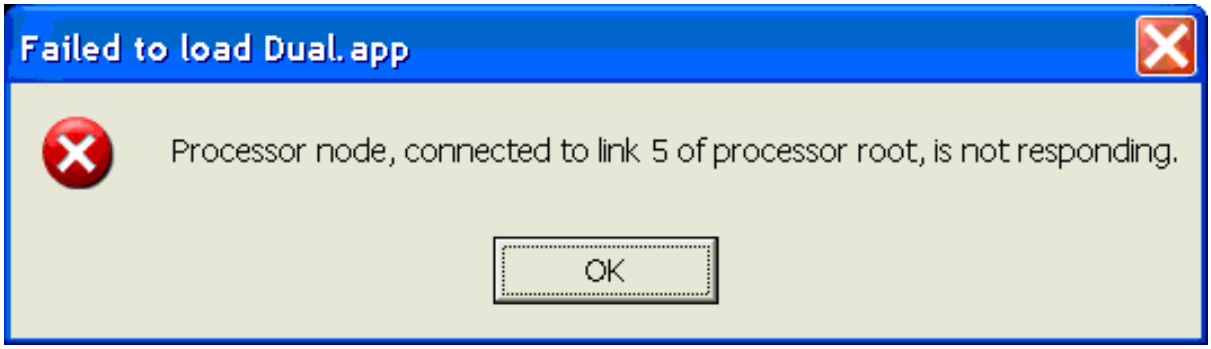
The server will attempt to send some words to the root processor describing the declared type of the root. If these words are not read within a few milliseconds, the server will report that the root processor cannot be accessed. You should check that you have selected the correct DSP board and that the link connection between the board and the host has been enabled.

The server now waits a few milliseconds for a reply from the root. If there is no reply, the server will report that the root processor is not responding. You should check that there is a TIM in the carrier board slot that is connected to the host link, and that a current version of the bootloader has been programmed into the flash ROM.

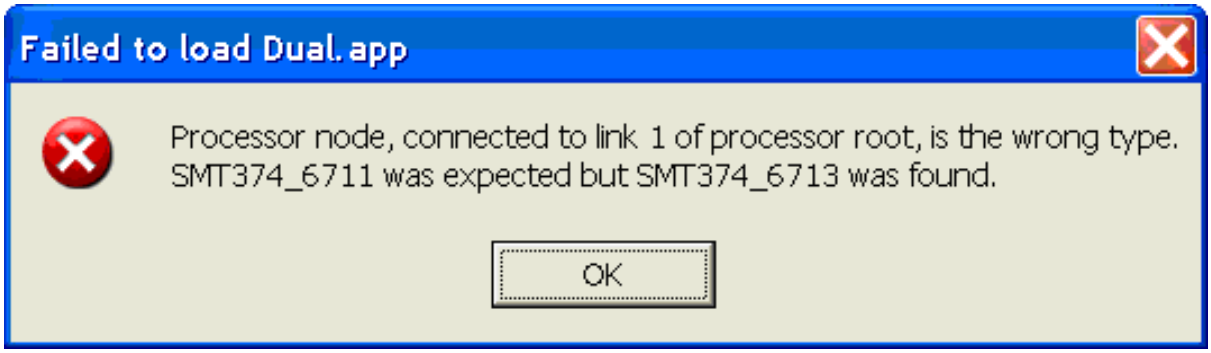
The response from the root will indicate whether or not its type is acceptable. If it is not, the server will report that the root is of the wrong type. You should check that the type of the root processor specified in the configuration file matches the actual type of the processor connected to the host link.



Finally the server sends the application down to the root which will distribute it to the rest of the network. Once everything has been sent, the server reads back a response from the root. A successful response means that the application has loaded correctly and it will start to execute. There are two possible failure responses:



One of the processors in the network is not responding. This could be the result of an incorrect link connection or the use of a TIM with an old bootloader. You should check that you have specified the correct WIRE connection between the named processors and that the non-responding processor has a current bootloader.



One of the processors in the network is not of the type expected. You should check that you have specified the correct WIRE connection between the named processors and that the actual processors connected in this way are of the correct types.

ROM Loading

The application will have been built as usual and then programmed into the Flash ROM by Sundance's SMT6001 utility. Following reset, the root processor's bootloader will detect that it is to boot from the ROM and will, in effect, construct a software comport link that will read the application data. Loading then proceeds as for Host Loading. Note that no load checking will be performed when loading from ROM.

Chapter 20. Sundance Digital Bus

From version 3.0 of Diamond you can use the Sundance Digital Bus (SDB) as an interprocessor link, much like a comport. This is the recommended way of using the SDB.

If you need more control, Diamond provides this alternative interface which supersedes the old SDB library that was available for some Sundance TIMs.

Warning

The SDB interface is not compatible with SDB links; you must choose one access method or the other as the two cannot co-exist.

You do not need to worry about the type of TIM you are using; the configurer will automatically select the correct SDB driver for you.

Terminology

Several Sundance TIMs have two SDBs, known as SDB-A and SDB-B. Newer modules can provide greater numbers of these devices where letters become cumbersome, so this document will identify SDBs by number: SDB0, SDB1, and so on.

Configuration

Before you can transfer data between two SDBs for the first time you need to set the connection into a consistent state, with one end being a transmitter and the other a receiver. The choice of which end is the transmitter is usually arbitrary; it does not depend on which end will actually send data first, as the SDBs will automatically switch direction once they have been correctly initialised.

You set the initial direction for the SDBs you are going to use with a qualifier on the PROCESSOR statement in your configuration file. There are two qualifiers: SDBTX= defines the transmitters and SDBRX= defines the receivers. The particular SDBs are specified by their index number. For example:

```
PROCESSOR A SMT361 SDBTX=0          ! SDB0 as a transmitter
PROCESSOR B SMT363 SDBRX=1,2        ! SDB1 and SDB2 as receivers
PROCESSOR C SMT335 SDBTX=1 SDBRX=0 ! SDB0 receive, SDB1 transmit
```

The configurer uses these attributes to select the device driver corresponding to your processor type and to initialise and synchronise the devices appropriately. If you do not specify either SDBRX or SDBTX, an SDB driver will not be loaded onto the processor.

For compatibility with older naming conventions, the configurer will also accept letters to identify each SDB (A=0, B=1, and so on).

Accessing an SDB

SMT_SDB_Claim [Stand-alone]

```
#include <SMT_SDB.h>
SMT_SDB *SMT_SDB_Claim(UINT32 which);
```

Each SDB is controlled by an interface of type SMT_SDB that is managed by the Diamond kernel. Before you can use an SDB you must call SMT_SDB_Claim to reserve it for your exclusive use and obtain its interface pointer. This function returns a NULL pointer if the SDB cannot be claimed.

SMT_SDB_Release [Stand-alone]

```
#include <SMT_SDB.h>
void SMT_SDB_Release(UINT32 which);
```

You can call this function to return an SDB to the kernel for use by other tasks. Most users should never need this function, as they will claim the SDB during initialisation and use it for the duration of the application. Following this call, interface pointers previously obtained using SMT_SDB_Claim with the same value of **`which`** may no longer be used.

SMT_SDB_Read [Stand-alone]

```
#include <SMT_SDB.h>
void SMT_SDB_Read(SMT_SDB *Si, UINT32 Bytes, void *Buffer);
```

Read the given number of bytes from the SDB identified by interface **`Si`** to the given buffer. **`Bytes`** must be a multiple of 4 and **`Buffer`** must be aligned on a 4-byte boundary. Si must have been claimed previously using `SMT_SDB_Claim`. This function will block (wait) until the data have been received and read. It is analogous to `chan_in_message`.

SMT_SDB_Write [Stand-alone]

```
#include <SMT_SDB.h>
void SMT_SDB_Write(SMT_SDB *Si, UINT32 Bytes, void *Buffer);
```

Write the given number of bytes to the SDB identified by interface **`Si`** from the given buffer. **`Bytes`** must be a multiple of 4 and **`Buffer`** must be aligned on a 4-byte boundary. **`Si`** must have been claimed previously using `SMT_SDB_Claim`. This function will block (wait) until the data have been transmitted. It is analogous to `chan_out_message`.

SMT_SDB_Control [Stand-alone]

```
#include <SMT_SDB.h>
void SMT_SDB_Control(SMT_SDB *Si, UINT32 Value);
```

This function sets the SDB Status register to the given **`Value`**. The initial setting of the status register will select:

SDB_CLROF	Clear outgoing FIFO
SDB_CLRIF	Clean incoming FIFO
SDB_CLK	Maximum clock speed
SDB_TRANS	(if set as SDBTX in config file)

The available control bits are briefly described in the header file, `<SMT_SDB.h>` .

Si must have been claimed previously using `SMT_SDB_Claim`.

Performance Issues

Warning

One end of an SDB attempting to write can block the other end from also being able to write at

the same time.

When possible, the drivers use DMA to transfer words between SDBs and memory. Unfortunately, the cache on newer TI processors is unable to maintain coherency between CPU and DMA accesses to external memory. The Diamond drivers work round this problem by a combination of flushing and invalidating the affected portions of the cache on each transfer, reducing the throughput by a minimum of 6%. You can take responsibility for addressing this potential problem yourself and stop the drivers from touching the cache by using the following function:

SMT_SDB_ProtectCache [Stand-alone]

```
#include <SMT_SDB.h>
void SMT_SDB_ProtectCache (SMT_SDB * Si, UINT32 Enable);
```

This function controls the cache management behaviour of an SDB. **Si** must have been claimed previously using SMT_SDB_Claim. Enable may be one of:

An SDB Example

```
// Sender.C
#include <SMT_SDB.h>
#include <stdio.h>
static int Values[8] = {1,3,5,7,9,8,6,2};

void main()
{
   int i;
   SMT_SDB *Sdb = SMT_SDB_Claim(0);
   int Changed[8];
   if (!Sdb) {
      printf("Cannot claim SDB0\n");  return;
   }

   SMT_SDB_Write(Sdb, sizeof(Values),  Values);
   SMT_SDB_Read (Sdb, sizeof(Changed), Changed);

   for (i=0; i<8; i++) {
      if (Changed[i]+Value[i]) {
         printf("Error: expected %d, received %d\n",
                -Value[i], Changed[i]);
      }
   }
   printf("Finished\n");
}

// Receiver.C
#include <SMT_SDB.h>

void main()
{
   int i;
   SMT_SDB *Sdb = SMT_SDB_Claim(0);
   int Data[8];
   if (!Sdb) return;
   SMT_SDB_Read (Sdb, sizeof(Data), Data);
   for (i=0; i<8; i++) Data[i] = -Data[i];
   SMT_SDB_Write(Sdb, sizeof(Data), Data);
}

! Configuration file
   PROCESSOR root SMT363 SDBTX=0
   PROCESSOR node SMT363 SDBRX=0
```

```
WIRE ? root[1] node[4]

TASK Sender    data=30K
TASK Receiver  data=30K

PLACE Sender   root
PLACE Receiver node
```

Chapter 21. Board Services

The board services module is a component of the Diamond kernel that is loaded into an application on-demand by the configurer. It provides an interface to services that are supported by Sundance carrier boards that use the V3 PCI bridge chip:

- Mailboxes
- Access to carrier-board SRAM
- The High-Speed Channel
- Packaged Services
- Faster Standard I/O
- File Transfers
- Access to PCI registers
- Access to PCI address space

These services can be used by tasks running on the processor that is in the first TIM slot of the carrier. They are supported on the host by the Sundance SMT6025 product. All of the functions here return an integer status code. Most do not require the use of the Diamond server on the host.

Accessing the Board Services Interface

You must include the header file SMT_BSI.h to access the board services module. The configurer will automatically arrange for the correct driver module to be added to the kernel on that processor.

```
#include <SMT_BSI.h>
```

Having done this, you have access to two functions:

OpenBoardServices [Stand-alone]

```
#include <SMT_BSI.h>
int OpenBoardServices(SC6xSmtI **I);
```

OpenBoardServices assigns an interface pointer to **`*I`**; this is then used in functions communicating with the board services module. The task that opens the board services module needs to have at least 350 bytes free on its heap. **`*I`** is set to be a null pointer if board services are not available or cannot be opened.

For example,

```
#include <SMT_BSI.h>
SC6xSmtI *BsI;
int status = OpenBoardServices(&BsI);
if (status != BS_OK) {
   printf("Cannot open board services (%d)\n", status);
   exit(1);
}
```

CloseBoardServices [Stand-alone]

```
#include <SMT_BSI.h>
int CloseBoardServices(void);
```

You call CloseBoardServices to close the interface once you have finished using it.

Using Board Services

Mailboxes

The carrier board's hardware provides sixteen, byte-wide mailboxes. The board services module organises these as two, bi-directional, 32-bit mailboxes, MB0 and MB1. The host can write to a mailbox and interrupt the DSP, which can then read the value. Similarly, the DSP can write to the host. Note that the High-Speed Channel claims MB1.

Smt_MbClaim [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_MbClaim(SC6xSmtI   *BsI,
                UINT32      Which,
                UINT32      Clients,
                SC6xSmtMb **I);
```

This function reserves a mailbox and sets ***I** to be an interface pointer for use in subsequent references to the mailbox. **BsI** is the interface to the board services module that was set by the call on OpenBoardServices. **Which** selects the mailbox you wish to claim and must be 0 or 1.

Clients specifies how incoming values on the mailbox are handled. The most significant bits of the value are interpreted as a client number in the range 0..Clients-1, and the whole value is used to satisfy a call to Smt_MbRead with that client number. The number of bits used will be the smallest number that can represent the number of clients: (int)ceil(log2 Clients). For example, if Clients is 1, no bits will be used and all values will satisfy a read; if Clients is 6, 3 bits will be used.

For example, assume that MB1 had been opened with the following call:

```
Smt_MbClaim(BsI, 1, 8, &Mb1);
```

A call Smt_MbRead(Mb1, 1, &x) will wait until the host writes a value to the mailbox. The call will be satisfied if the host writes a value such as 23456789_{16}, because the most significant 3 bits (23=8) have the value 1. The call will then set x to the value 23456789_{16}. At the same time, another thread could be waiting on a call: Smt_MbRead(MB1, 4), but this call would not be satisfied by the host write as the client number 4 does not match the most significant 3 bits of the value sent.

Smt_MbRelease [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_MbRelease(SC6xSmtI *BsI,
                  UINT32    Which);
```

Release a claimed mailbox and make it available for use elsewhere. Following this call, the interface pointer returned by the corresponding call to Smt_MbClaim is no longer valid.

Smt_MbRead [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_MbRead(SC6xSmtMb *MbI,
               UINT32     Client,
               UINT32    *Arg);
```

Wait for the host to write a suitable value to the mailbox, then read that value into ***Arg**. If the value had already been written before this function is called, the function will not wait but will store the value in *Arg immediately. The function will only accept values where **Client** matches the bottom bits of the value, as determined by the parameter **Clients** in the call to Smt_MbClaim. **Client**

must be in the range 0..Clients-1. Note that Client=0 when Clients was 1 will match all values sent from the host.

You must not have more than one call of Smt_MbRead with the same value of Client active at any time.

Smt_MbWrite [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_MbWrite(SC6xSmtMb *MbI,
                UINT32     Value);
```

Write **Value** to the mailbox **Mb**. The call will not return until the value has been read from the mailbox by the host.

Carrier-board SRAM

The carrier board includes 1MB of SRAM that can be directly addressed from the host PC. The board services module provides functions to transfer data between this SRAM and the root DSP's memory. You will usually need to indicate when the transfer may be done (when data has been written to SRAM or SRAM has space available for writing) by using another channel to the host, commonly the host comport or Mailboxes. Note that the High-Speed Channel uses all of the SRAM to communicate address information.

In the following functions, Offset is a byte offset from the start of the SRAM, Bytes indicates how many bytes of data are to be moved (must be a multiple of 4), and Buffer points to the first word of DSP memory that will provide or receive the data.

Smt_SramRead [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_SramRead(SC6xSmtI *BsI,
                 UINT32    Offset,
                 UINT32    Bytes,
                 void     *Buffer);
```

Read from the SRAM into the given buffer.

Smt_SramWrite [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_SramWrite(SC6xSmtI *BsI,
                  UINT32    Offset,
                  UINT32    Bytes,
                  void     *Buffer);
```

Write the values in the given buffer to the SRAM.

The High Speed Channels

The High-Speed Channels (HSC) provide routes to the host that can be used to transfer data between the PC and the DSP at rates that are considerably greater than can be achieved using the host comport link.

The HSC provides eight channels using a mechanism based on three features of Sundance carrier boards:

1. Mailboxes, which allow interrupt-driven communication of single 32-bit words between the host and the

DSP.

2. SRAM, which is an area of shared memory that can be directly accessed by the PC and indirectly accessed by the DSP; and

3. PCI access, which allows data transmission between the DSP memory and locked-down memory area on the host. In order to do this, the DSP needs to have a definition of where the PC has locked the memory. The PC provides this information as a *Memory Descriptor List* (MDL) that is written to part of the SRAM. The MDL for a channel is initialised on an `OpenPci` function call and will not be altered until the memory is released by a `ClosePci` call. Between these calls, the DSP is free to cache information about the host's memory.

Note

References to a particular high-speed channel must be sequential. You may not have one thread reading from a channel while another thread is writing to the same channel. It is safe to access different channels at the same time.

Mailbox Usage

The HSC claims mailbox 1. The DSP will send a mailbox value to the host to request some action; the Host eventually sends a mailbox value back as a reply to the DSP. Each 32-bit value is interpreted as follows:

```
typedef struct {
   UINT32 Data     :25;   // data value
   UINT32 Fn       : 4;   // function code
   UINT32 Channel  : 3;   // channel selector
} HSC_WORD;

#define HscChannel(v) (*(HSC_WORD *)&(v)).Channel
#define HscFn(v)      (*(HSC_WORD *)&(v)).Fn
#define HscData(v)    (*(HSC_WORD *)&(v)).Data
```

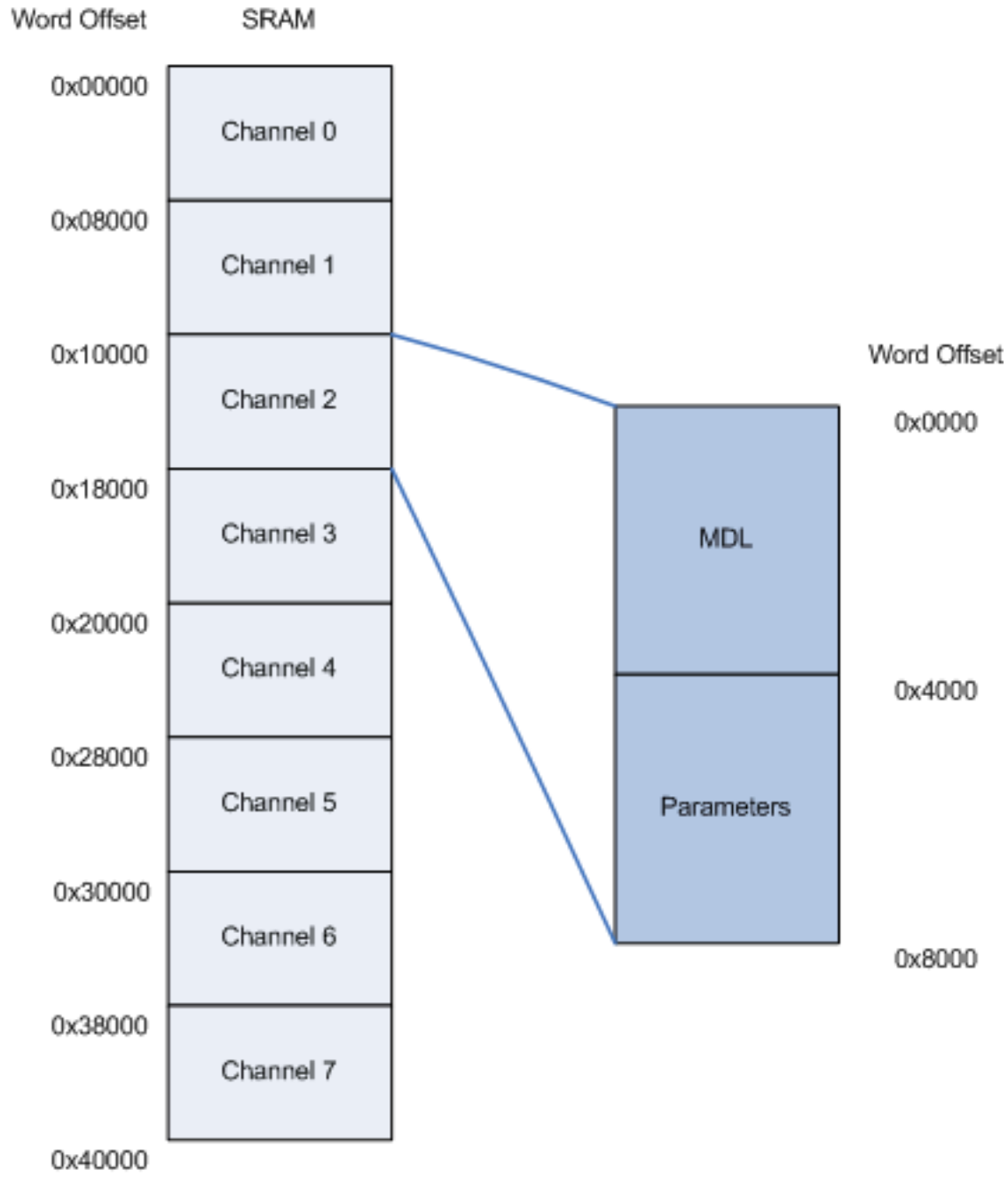
`Channel` selects one of the eight available channels.

`Fn` holds a function code that the DSP uses to inform the host what is to be done. The host may only set **`Fn`** to the values 0 (OK) or 1 (Error).

`Data` holds a parameter value for the selected function.

SRAM

The SMT310Q's 1MB of SRAM is used by the high-speed channels and cannot be used at the same time for any other purpose. It is broken down into eight equal areas, one for each of the eight high speed channels, allocated as follows:



The SRAM can be accessed as bytes by the PC, but only as 32-bit words by the DSP.

DSP Function Codes

fn	Meaning	Host interpretation
0	OK	No operation
1	Error	Error number (>0) in "data". Error 0 is undefined.
2	OpenHandler	User handler dll (name in SRAM) on this channel
3	CloseHandler	Close handler and revert to default handler
4	SramToHost	Take "data" bytes [a] from SRAM parameter area
5	HostToSram	Put up to "data" bytes [a] into the SRAM parameter area
6	OpenPci	Claim and lock host memory. Data gives the size of the area in bytes [a].

fn	Meaning	Host interpretation
7	ClosePci	Release any locked memory
8	PciToHost	Take "data" bytes [a] from PCI buffer (needs MDL)
9	HostToPci	Put up to "data" bytes [a] into PCI buffer (needs MDL)
10..15		Handler-specific functions

[a] All transfer and memory sizes are given in bytes, but they will be rounded this up and a whole number of words will be moved. User code should ensure there is enough space for this to be done safely.

Host Replies

The host can only ever reply with fn=0 (OK) or fn=1 (ERROR).

If the reply has fn=1, Data holds a non-zero error code. If fn=0, the value of Data means:

	Replying to	Data contains
0	OK	0
1	Error	0
2	OpenHandler	0
3	CloseHandler	0
4	SramToHost	number of bytes taken from SRAM
5	HostToSram	maximum number of bytes put in SRAM
6	OpenPci	maximum transfer size in bytes
7	ClosePci	0
8	PciToHost	number of bytes taken from PCI buffer
9	HostToPci	maximum number of bytes put in PCI buffer
10..15		handler-specific

Protocol

All host transactions are of the following form:

1. Read mailbox control word from DSP;
2. Read any parameters from the parameter area;
3. Perform required actions;
4. Write any results to the parameter area;
5. Write mailbox reply control word to DSP.

All DSP transactions are of the following form:

1. Write any parameters or data to the parameter area or PCI buffer;
2. Send mailbox control word to host;
3. Read mailbox reply control word;
4. Transfer any data out of PCI buffer or parameter area.

DSP Functions

In the following functions, HSC is the type of the interface needed to access the High Speed Channel.

```
int Smt_HscControl  (HSC *I, UINT32 Chan, UINT32 Fn, UINT32 Data);
int Smt_HscPciRead  (HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);
int Smt_HscPciWrite (HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);
```

```
int Smt_HscReadArgs (HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);
int Smt_HscWriteArgs(HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);

int Smt_HscControl(HSC *I, UINT32 Channel, UINT32 Fn, UINT32 Data)
{
   UINT32 V = 0;
   HscChannel(V) = Channel;
   HscFn(V)      = Fn;
   HscData(V)    = Data;
   MboxWrite(V);
   V = MboxRead();
   if (HscFn(V) != OK) return (V&0x1FFFFFFF) | 0x80000000;
   return HscData(V);
}

int Smt_HscPciRead(HSC *I, UINT32 Channel, UINT32 Bytes, void *Buf)
{
   int n = Smt_HscControl(I, Channel, HostToPci, Bytes);
   if (n > 0) {
      n = MIN(n Bytes);
      MoveFromHost(n, Buf);            // from Host buffer to Buf
   }
   return n;
}

int Smt_HscPciWrite(HSC *I, UINT32 Channel, UINT32 Bytes, void *Buf)
{
   int r = MoveToHost(Bytes, Buf);      // from Buf to Host buffer
   if (r < 0) return r;                 // error ?
   return Smt_HscControl(I, Channel, HostToPci, n);
}
```

Smt_HscReadArgs [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_HscReadArgs(HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);
```

Move the given number of bytes from the parameter area for the given channel to the given buffer.

Smt_HscWriteArgs [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_HscWriteArgs(HSC *I, UINT32 Chan, UINT32 Bytes, void *Buf);
```

Move the given number of bytes to the parameter area for the given channel from the given buffer.

Smt_HscInit

```
#include <SMT_BSI.h>
int Smt_HscInit ( SC6xSmtI * BsI);
```

This function makes the High-Speed Channel available for use. It attempts to claim mailbox 1 (MB1), which it will use to coordinate transfer requests with the host. You need at least 2,500 bytes available from the heap when this function is called.

DSP Example

```
#define CHANNEL 5                                   // using channel 5
```

```
#define OpenFile  11                                  // handler-specific
#define CloseFile 12                                  // handler-specific

int ReadFile(char *name)
{
   int i, n, Max;
   struct {
      int Code;
      int Arg;
      char Name[256];
   } P;

   P->Code = 0;                                       // unused
   P->Arg  = 0;                                       // unused
   strcpy(P->Name, "HscFile.dll");                    // handler name
   HscPutArgs(I, CHANNEL, sizeof(P), &P);
   n = Smt_HscControl(I, CHANNEL, OpenHandler, 0);
   if (n < 0) return n;                               // no handler

   P->Code = 1;                                       // Open for input
   P->Arg  = 0;                                       // unused
   strcpy(P->Name, name);                             // Filename
   HscPutArgs(I, CHANNEL, sizeof(P), &P);
   n = Smt_HscControl(I, CHANNEL, OpenFile, 4096);// 4KB buffer
   if (n > 0) {
      for (i=0; i<4; i++) {                           // 4 x 128 bytes?
         n = HscRead(I, CHANNEL, 128, Buffer);
         if (n==0) break;
         Process(n, Buffer);
      }
      Smt_HscControl(I, CHANNEL, CloseFile, 0);   // Close
   }
   Smt_HscControl(I, CHANNEL, CloseHandler, 0);
   return n;
}
```

This would result in the following transactions on channel 5, assuming the file contains 70 words:

Host		**DSP**
	<==	OpenHandler
OK	==><	==
	<==	OpenFile, Data=4096
OK, Data=4096	==><	==
	<==	GivePci, Data=128
OK, Data=128	==><	==
	<==	GivePci, Data=128
OK, Data=128	==><	==
	<==	GivePci, Data=128
OK, Data=24	==><	==
	<==	GivePci, Data=128
OK, Data=0	==><	==
	<==	CloseFile
OK	==><	==
	<==	CloseHandler
OK	==><	==

Accessing PCI Registers

Smt_PciRegRead [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_PciRegRead(SC6xSmtI *BsI,
                   UINT32    Woffset,
                   UINT32   *Value);
```

*Value is set to the contents of the PCI register located at the given word offset, Woffset, from the PCI base. The board service module should be locked when you issue this call. See Smt_Claim.

Smt_PciRegWrite [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_PciRegWrite(SC6xSmtI *BsI,
                    UINT32    Woffset,
                    UINT32    Value);
```

Write Value to a PCI register located at the given word offset, Woffset, from the PCI base. The board service module should be locked when you issue this call. See Smt_Claim.

PCI Access

The board services module provides functions to transfer data between memory in the PCI address space and the root DSP's memory. Misuse of these functions can easily cause the host PC to crash.

In the following functions, PciAddr is an address in the PCI address space, Bytes indicates how many bytes of data are to be moved (must be a multiple of 4), and Buffer points to the DSP memory that will provide or receive the data.

Smt_PciRead [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_PciRead(SC6xSmtI *BsI,
                UINT32    PciAddr,
                UINT32    Bytes,
                void     *Buffer);
```

Read the given number of bytes from the given PCI address into the DSP's buffer.

Smt_PciWrite [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_PciWrite(SC6xSmtI *BsI,
                 UINT32    PciAddr,
                 UINT32    Bytes,
                 void     *Buffer);
```

Write the given number of bytes from the DSP's buffer to the given PCI address.

The Global Bus

The functions discussed above will be sufficient for most users. Occasionally it may be necessary to access devices directly on the global bus. You can perform such accesses using Smt_GbRead and Smt_GbWrite. In

these functions, GbAddr is the address issued to the global bus, while Bytes and Buffer define the memory to be used in the DSP. Bytes must be a multiple of 4 and Buffer must be aligned on a 4 byte boundary. Control is the value to be used in the global bus control register for the transfer. Details of this can be found in the User Guide for your TIM.

Smt_GbRead [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_GbRead(SC6xSmtI *BsI,
               UINT32    GbAddr,
               UINT32    Bytes,
               void     *Buffer);
               UINT32    Control);
```

Read from the global bus.

Smt_GbWrite [Stand-alone]

```
#include <SMT_BSI.h>
int Smt_GbRead(SC6xSmtI *BsI,
               UINT32    GbAddr,
               UINT32    Bytes,
               void     *Buffer);
               UINT32    Control);
```

Write to the global bus

Packaged Services

Faster Standard I/O

At any time, a task on the root can make a single function call that will switch the host link away from the host comport to the HSC. From then on, all host references will use the HSC. In particular, all the standard library calls (fopen, fread, fwrite,…) will continue to work as before, but at a potentially greater rate. Once the HSC has been selected in this way it will not be possible to switch back to the host comport and you will not be able to use the HSC for any other purpose. This service requires the Diamond server to be running on the host.

select_fast_host_io [Stand-alone]

```
#include <SMT_FHIO.h>
int select_fast_host_io ( void);
```

This function switches the host link from a comport to the high-speed channel. It should be called only once from any host task. It is only worthwhile doing this if you transfer large amounts of data between the host and the root processor.

The return value is zero for success. Failure is indicated by a non-zero return code, and is most commonly the result of using old or inappropriate firmware.

Once the host link has been switched to the HSC it cannot be switched back to the comport.

For example:

```
// Examples\General\FhostIO.C

#include <stdio.h>

#include <SMT_FHIO.h>
```

```
main()
{
   int n;
   char *where = "comport";
   for (n=0; n<200; n++) {
      printf("This goes across the %s: %d\n", where, n);
      if (n==99) {
         int e = select_fast_host_io();
         if (e==0) {
            where = "HSC";
         } else {
            printf("Cannot switch to HSC. Error=%d\n", e);
         }
      }
   }
}
```

File Transfers

There is an HscFile library, available from Sundance, that implements file transfers across the HSC. Its functionality is similar to fread and fwrite, but is sacrifices the flexibility of those functions in order to gain increased performance.

Performance

Although the HSC can transfer data quickly, there is a considerable overhead in setting up and starting each individual transfer. This is not significant when you are moving large volumes of data, but can dominate the time taken to handle small transfers.

Status Codes

The following values are returned by the board services functions:

Macro	Value	Meaning
BS_OK	0	Success
BS_NOT_OPEN	-1	Services not opened yet
BS_DIRECTION	-2	Simultaneous read and write
BS_NO_DMA	-3	cannot find DMA interface
BS_NO_EXTINT	-4	cannot find external interrupt interface
	-5	
BS_NO_MEMORY	-6	cannot claim working memory (heap)
BS_NO_DATA	-7	Host has given a zero length memory area
BS_MBOX1_BUSY	-8	Mailbox 1 already claimed
BS_BAD_FIRMWARE_CTR	-9	Firmware error: Contact Sundance
BS_BAD_FIRMWARE_ADDR	-10	Firmware error: Contact Sundance

Chapter 22. Sundance TIMs

Sundance C6000 processors are build around a range of modules called TIMs. Each TIM uses an FPGA to implement the Sundance peripherals such as comports and SDBs.

Most Diamond applications do not need to know anything about the FPGA, but occasionally it may be necessary to be able to access it. The header file `<smt_fpga.h>` provides definitions of the standard registers that are used to control FPGA resources. In particular, it contains a function **`SMT_FpgaBase`** that computes the base address used to access FPGA registers.

SMT_FpgaBase [Stand-alone]

```
#include <smt_fpga.h>
unsigned int SMT_FpgaBase(void);
```

This function returns the address of the start of the FPGA in the processor's address space. It is used in conjunction with the register offsets also defined in `<smt_fpga.h>` for low-level access to Sundance devices.

The base address will vary from TIM to TIM, so the header file normally only defines offsets that can be added to the actual base address of the FPGA to generate the address of the registers.

For example, the LED register can be found as follows:

```
#include <smt_fpga.h>      // only defines offsets

unsigned int *LED;

main()
{
   LED = (unsigned int *)(SMT_FpgaBase() + LED_OFFSET);

   *LED = 7;  // turn on 3 LEDs
}
```

If your code does not need to be general and you know the FPGA address for the TIM on which it will always run, you can condition the header file by defining **`___TIM`** to be that address. Now when you include `<smt_fpga.h>` it will provide definitions for all the registers, as shown in the following example which is identical to the one above except that it will only run on an SMT361.

```
#define ___TIM 0x90000000  // define base of FPGA for SMT361
                           // note the three underlines.
#include <SMT_FPGA.h>      // now defines LED for you

main()
{
   *LED = 7;               // turn on 3 LEDs
}
```

Chapter 23. ROM

Sundance TIMs have a Flash ROM into which you can place areas of data. The Flash appears as a read-only area of addressable memory; you can only access it in units of 32 bits (words). The most common use of these areas is to hold a program that is loaded and executed automatically whenever the TIM is reset. It is also possible to include blocks of data for use by any running applications. Sundance's documentation for the SMT6001 gives details of the Flash ROMs, how they may be programmed, and the format of the data they can hold.

Note

No special action is needed to allow stand-alone Diamond applications to be programmed into the ROM. [a]

[a] Previous versions of Diamond have required special options to be selected when configuring applications for ROM; this is now no longer necessary.

SMT_RomBase [Stand-alone]

```
unsigned int *SMT_RomBase(void);
```

This function returns the address of the start of the flash ROM. It is normally only used in conjunction with the Sundance Flash ROM library.

Chapter 24. APP2COFF

App2Coff is a utility that will take a Diamond **.app** file and convert it into a single COFF **.out** file that can be used with Code Composer to load the application down a JTAG connection to the root processor of a network. When the root starts to execute, it loads the rest of the network over the interconnecting comports. This utility is intended only for use in exceptional circumstances; the Diamond server, WS3L, provides the preferred way of loading and executing applications.

Constraints

1. The application must be stand-alone, that is, it must not attempt to communicate with a host server;
2. All tasks in the application must be linked with the stand-alone library (usually using **3L ta**);
3. The application must be built using the configurer's **-a** switch. Note that this will also prevent the configurer from including load checking information.
4. There must be enough uninitialised memory available on the root processor for all the data of any non-root processors used by the application. This memory can be subsequently re-used for data areas on the root (specifically, stack and heap space).
5. SMT335 and SMT375 processors are not supported.

Using App2Coff

First, build your Diamond application in the usual way, making sure that all the tasks are linked against the stand-alone library and that you use the **-a** switch when configuring. For example, assuming the configuration file `myprog.cfg` contains:

```
PROCESSOR root SMT376_6711_128
PROCESSOR node SMT376_6711_256
WIRE       ?    root[1] node[4]
TASK            main  data=1M   ins=1 outs=1
TASK            other data=256K ins=1 outs=1
CONNECT    ?    main[0]  other[0]
CONNECT    ?    other[0] main[0]
PLACE           main  root
PLACE           other node
```

You can build an application like this:

```
► 3L c   main.c
► 3L ta  main
► 3L c   other.c
► 3L ta  other
► 3L a   myprog -a
```

You then use App2Coff, giving the base name of the application file (the filename without the **.app** extension) and the type of the root processor. This is the same as the processor type you give for the root processor in the configuration file.

```
# App2Coff file processor-type
```

Continuing the previous example, you could build the COFF file `myprog.out` from the application

`myprog.app` with the command:

```
► App2Coff myprog SMT376_6711_128
```

Loading the application

When loading a network with an application converted to COFF by App2Coff, you should follow the following sequence:

1. Reset all of the processors in your network. This step would normally be done for you by the Diamond server.
2. Select Debug/Run Free for all of the processors. This is important as it allows the processors to initialise their Sundance I/O devices, such as the comports.
3. Load the **`.out`** file into the root processor.
4. Let the root processor Run Free.

Note that all four of these steps will be needed if you wish to start the application running again. You must not use Debug/Restart.

App2Coff Error Messages

cannot access filename

The given file cannot be found.

cannot create filename

The output file cannot be created. Check that you have space on the output device and that the output file is neither in use nor write-protected.

cannot find memory for object.

App2Coff needs to locate memory areas to hold a small loader and data for the application (mainly the information needed to load non-root processors). This error indicates that there isn't a contiguous lump of unallocated memory on the root processor large enough to hold the given object.

corrupt application file filename

This is usually the result of specifying a file that is not a Diamond application or one that has been corrupted in some way.

error reading filename

This is usually the result of specifying a file that is not a Diamond application or one that has been corrupted in some way.

file is too large to be processed

This is usually the result of specifying a file that is not a Diamond application.

filename references unavailable memory on processor: nnn bytes at address.

Your application (filename) has a reference to some memory that is not available on the given processor. This is most commonly the result of specifying a processor type in the App2Coff command that is different from the type given for the root in the configuration file.

internal error/nnn

This indicates that an internal consistency check has failed. Rebuild your application and try again. If the error persists, please contact 3L.

unknown processor type xxx

You have given a processor type that does not correspond to any of the TIMs that App2Coff supports. Check the spelling of the processor type.

Chapter 25. The Worm

The worm is a Diamond utility that you can use to explore certain DSP networks dynamically. It detects all the processors that are in your network and determines the communication port connections amongst them. The worm may also be used either to create the hardware portion of a configuration file (PROCESSOR and WIRE statements) or to verify that a given configuration file accurately describes the hardware and connections you actually have.

Accepted Networks

The current worm can only operate on homogeneous networks: networks where all the processors are of compatible type and are connected by communication ports. There are currently four worms available:

WORM3x5.APP	Works on networks built from SMT335(E) and SMT375(E) TIMs.
WORM361.APP	Works on networks built from SMT361 TIMs
WORM363.APP	Works on networks built from SMT363 TIMs
WORM376.APP	Works on networks built from all flavours of SMT376 TIMs

Any links connected to devices that are not SMT TIMs of the appropriate type will be reported as being unconnected. The worm is likely to hang or report the wrong structure if it is used on the wrong type of network.

The rest of this chapter will assume a network built from STM3x5 processors.

Starting the Worm

The worm is a normal Diamond application and is started using the Diamond server, either using the Windows interface or from the DOS command line:

```
WS3L wormxxx switches
```

For example:

```
► WS3l worm3x5 /c
```

The switches are a sequence of items used to control the operation of the worm. When the worm is started using the Windows interface, the switches must be specified in the command line box.

Switches

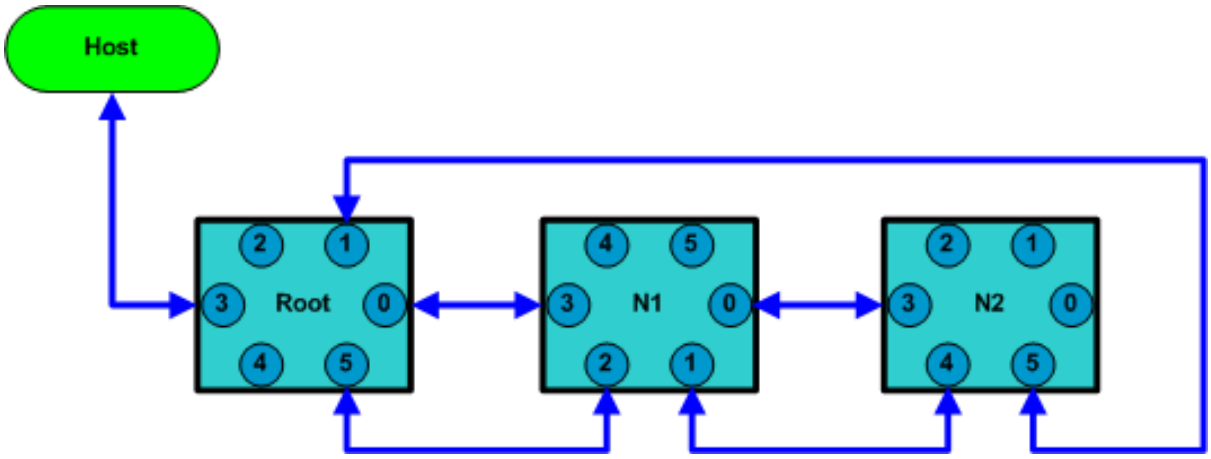
The worm’s switches start with "/" or "-". They are not case-sensitive.

-c	Configuration	The worm determines the structure of your network and outputs PROCESSOR and WIRE statements suitable for the hardware description part of a configuration file. The processors will be named ROOT, N1, N2, and so on. The output is sent to the standard output stream. Note that the Worm is currently unable to distinguish different processors from the same family (e.g., SMT376_6711_128 and SMT376_6711_256). It will give the processors a generic name that you will need to change to match the actual TIMs you have.

-h	Help	A brief description of the worm command line and its switches is sent to the standard output stream.
-v filename	Verify	The worm compares the actual structure of your network with the structure described in the configuration file "filename". Only PROCESSOR and WIRE statements in the configuration file are interpreted. If the actual and described structures are equivalent, the worm stops with a zero return code; otherwise it outputs a message and stops with a non-zero return code. The structures are deemed equivalent if the described structure in "filename" is a subset of the actual structure found. In other words, the actual network may have more processors and connections than the network described in "filename".

Worm Output

As an example, consider the following network:



When run with no switches on this network, the worm will generate the following output:

```
Processor link#0 link#1 link#2 link#3 link#4 link#5
ROOT:       N1:3     N2:5     ****:*   HOST     ****:*   N1:2
N1:         N2:3     ****:*   ROOT:5   ROOT:0   N2:1     ****:*
N2:         ****:*   N1:4     ****:*   N1:0     ****:*   ROOT:1
```

Where:

N1:3	means link #3 of processor N1
HOST	means the connection to the host PC
****:*	means no connection to another processor
(blank)	means there is no such comport link

When the "-c" switch is used, the worm will output part of a configuration file like this:

```
PROCESSOR ROOT SMT335
PROCESSOR   N1 SMT335
PROCESSOR   N2 SMT335

WIRE ? ROOT[0]   N1[3]
```

```
WIRE ? ROOT[1]   N2[5]
WIRE ? ROOT[5]   N1[2]
WIRE ?   N1[0]   N2[3]
WIRE ?   N1[4]   N2[1]
```

Part IV. Bibliography

Table of Contents

Chapter 26. Bibliography

[ANSI-C] .

American National Standard for Information Systems—Programming Language—C., 1990. X3.159-1989.

[Hoare] .

Communicating Sequential Processes. Prentice-Hall, 1985. ISBN 0-131-53271-5.

[Lister] .

Fundamentals of Operating Systems. Macmillan Press, 1979. ISBN 0-333-27287-0.

[Tanenbaum] .

Operating Systems: Design and Implementation. Prentice-Hall, 1987. ISBN 0-13-637331-3.

[Metalanguage] .

British Standard BS6154 : 1982: Method of Defining Syntactic Metalanguage, 1981. ISBN 0-580-12530-0.

[Scowen] .

An Introduction and Handbook for the Standard Syntactic Metalanguage. National Physical Laboratory Report DITC 19/83, February 1983.

[SPRU509] .

Code Composer Studio Getting Started Guide

[C Compiler] .

TMS320C6x Optimizing C Compiler User's Guide., February 1998. SPRU187C.

[Assembler] .

TMS320C6x Assembly Language Tools User's Guide. Texas Instruments Inc., February 1998. SPRU186C.

[Guide] .

TMS320C62x/67x Programmer's Guide. Texas Instruments Inc., February 1998. SPRU198B.

[CPU] .

TMS320C62xx CPU and Instruction Set Reference Guide. Texas Instruments Inc., July 1997. SPRU189B.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W