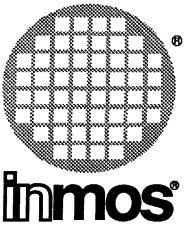


3L PARALLEL C USER GUIDE

72 TDS 179 00

February 1989

72 TDS 179 00

Contents

Contents overview

The appendices

1 How to use the manual

Intended audience

This *User Guide* accompanies 3L's Parallel C product, and is intended for anyone who wants to use Parallel C to program a transputer system, whether writing a conventional sequential program or using the full support for concurrency which the transputer processor has to offer.

1.1 How to use the manual

There are three main divisions within this document, as follows:

- User guide
- Reference manual
- Appendices

Each of the sections is described briefly below.

1.2 User guide

The user guide introduces you to the operation of the compiler and the other tools supplied with Parallel C. In particular, there are tutorial sections explaining parallelism on the transputer and the way in which this can be accessed from Parallel C programs.

1.3 Reference manual

The reference manual contains the detailed technical information which you will require to write sophisticated applications for the transputer using Parallel C.

1.4 Appendices

The appendices at the end of this manual contain supplementary information in a condensed form, such as tables of transputer assembly language mnemonics.

Further reading

This *User Guide* does not attempt to teach the C language itself; rather, reference should be made to one of the many introductory texts available. The first — and still one of the best — books about C is the original book describing the language. This is *The C Programming Language*[1], by Kernighan and Ritchie.

In a similar way, the reader is assumed to be reasonably familiar with the operating system of the host computer being used. For personal computers made by IBM, this will usually be PC-DOS, which is supplied with a manual called *Disk Operating System Reference*[2]. For compatible machines made by other manufacturers, the operating system will usually be MS-DOS, described in *Microsoft MS-DOS User's Reference*[3]. These two operating systems are largely compatible, and their documentation is very similar.

References to these and other documents mentioned in this manual are collected in a bibliography, which can be found on page 259.

1.5 Host operating system dependencies

Operating system dependencies are as far as possible made invisible to the user. The few differences are summarized below.

Command line syntax

The major difference between different host implementations is the option prefix character. For UNIX based toolsets the prefix character is the hyphen '–'; for all other toolsets it is the forward slash character '/'.

This manual uses the '–' character in all examples where the tools are invoked from the host operating system.

Directories and files

Directories and pathnames are treated in a host dependent manner, whereas filenames are independent of the host, with certain restrictions. As long as the directory names are legal for the host operating system, they can also be treated as host independent.

A directory path searching mechanism is implemented within the compiler, and full pathnames need not be given.

The user guide

72 TDS 179 00

2 Programming single transputers

2.1 Outline procedure

To create a program to run on a single transputer you will need to take the following steps:

1 Compile the C source texts using the C compiler.

2 Link the object files with the C run-time library using the linker.

3 Convert the linked image to an executable file using the bootstrap tool.

4 Load the program for execution on the transputer board using the server.

It is often convenient to collect the object files for commonly used functions into a single library file. This can be done using the librarian.

2.2 A simple example

The following example shows how to build and run the '**hello world**' program for a T414 transputer.

The source of the **hello world** program is held in the file **hello.c**. To compile this source for a T414 transputer type:

```
t4c hello
```

As no filename extension is given the compiler will add a **.c** extension automatically. If the source file contains no errors, an object file **hello.bin** will be produced, otherwise error messages will be written to the standard output stream.

The object file must now be linked with the C run-time library and converted to an executable file. The sequence of commands required to perform this is as follows:

```
ilink mainent.c4x crtl.lib hello.bin -o hello.c4x
iboot hello.c4x
```

To run the program type:

```
iserver -sb hello.b4x
```

The **iserver** loads the program **hello.b4x** to the transputer where it executes automatically.

To build a program for a T800 transputer a similar sequence of commands shown below must be used.

```
t8c hello
ilink mainent.c8x crtl.lib hello.bin -o hello.c8x
iboot hello.c8x
iserver -sb hello.b8x
```

2.3 A more complex example

For larger programs it is good practice to build them from a number of separately compiled C source texts. The following example shows how to build a program for a T414 from the two C source texts **f1.c** and **f2.c**. The example can easily be generalised for more source files. First compile each source text separately:

```
t4c f1
t4c f2
```

These commands will create the object files **f1.bin** and **f2.bin**. The object files must now be linked with the C start-up routine and the C run-time library:

```
ilink mainent.c4x f1.bin f2.bin crtl.lib -o main.c4x
```

The C start-up routine (**mainent.c4x**) and C run-time library (**crtl.lib**) can be found in the installation directory defined in the delivery manual. If the compiler search path is set up as described in the delivery manual the linker will find these without needing an explicit directory specifier. Note that **.bin** extensions must be stated explicitly.

The **-o main.c4x** linker option explicitly names the output file. Before the program is run the linked image must be converted to an executable file:

```
iboot main.c4x
```

The bootstrap program produces the executable file **main.b4x** as output which can be loaded by the server to run on your transputer board:

```
iserver -sb main.b4x
```

The same program can be built for a T800 transputer using the similar sequence of commands below:

```
t8c f1
t8c f2
ilink mainent.c8x f1.bin f2.bin crtl.lib -o main.c8x
iboot main.c8x
iserver -sb main.b8x
```

2.4 Indirect linker files

When a program consists of many separately compiled units it is recommended that the linker be invoked using an indirect file.

By creating a file **main.l4x** containing:

```
mainent.c4x
f1.bin
f2.bin
crtl.lib
-o main.c4x
```

the previous example could be linked with the command line:

```
ilink -f main.l4x
```

where the **-f** option instructs the linker to take its command line from the indirect file **main.l4x**.

2.5 Libraries

Collecting a group of object files into a library has a number of advantages:

1 A single file is more convenient than a collection, especially if it is to be shared and copied between a team of developers.

2 Object code for different transputer types may be collected into a single library, from which the linker will select the correct version.

3 The linker will automatically select only the object files required for the program from a library.

The example below shows a graphics library being built from a core graphics module and two device driver modules. The library is then linked in the ordinary way with a user program. Indirect files are used to simplify the required librarian and linker commands.

Contents of **graflib.lbb** are:

```
core.bin
tek.bin
hp.bin
```

The following command is used to build the library:

```
ilibr -f graflib.lbb -o graflib.lib
```

Contents of **myprog.14x** are:

```
mainent.c4x
myprog.bin
crtl.lib
graflib.lib
```

The following commands are used to link the program and produce the bootable file:

```
ilink -f myprog.14x -o myprog.c4x
iboot myprog.c4x
```

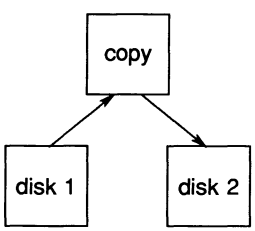
3 Introduction to Parallel C

This chapter aims to help you become familiar with Parallel C and its terminology. If you know occam, or if you have read a lot about the transputer, then you will already be familiar with the ideas on which Parallel C is based. If not, don't worry; the ideas are quite simple. They are explained in outline here, and again in more detail in the chapters which follow.

3.1 Abstract model

The treatment of parallel processing in transputer systems is based on the idea of *communicating sequential processes.* In this model, a computing system is a collection of concurrently active sequential processes which can *only* communicate with each other over *channels.* A channel connects exactly 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. Each process can have any number of input and output channels, but note that the channels in a system are fixed; new channels cannot be created during its operation.

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. 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 just a 'black box' connected to the outside world only by its channels, the actual implementation of any individual process is not important. A process could be implemented in hardware or software or could be a complex system, itself consisting of a number of communicating processes.

3.2 Hardware realisation

The transputer was designed to be used as a component in concurrent systems of exactly the sort described in the previous section. Each transputer *processor* has four INMOS *links,* to connect it with other transputers. Each link has two channels, one in each direction. These hardware channels behave exactly like the abstract channels discussed above; they provide synchronised, unidirectional communication.

Arbitrary *networks* of transputers can be constructed simply by connecting their links together with ordinary *wires*, the only limitation being that each processor cannot be directly connected to more than four others.

At this level, a transputer can therefore be viewed as a single process in a multi-transputer system. However, it is also possible for any number of concurrent processes to be run on an individual transputer. Any word in the transputer's memory may be used as a channel to connect one internal process to another. The address of such a channel word is used to identify it to the transputer instructions (and Parallel C functions) which send or receive messages. The contents of the word are used by the hardware to synchronise sending and receiving processes.

From a program's point of view, these internal channels and the hardware link channels are identical. The same instructions (or Parallel C functions) are used to send and receive messages on both. Hardware link channels are identified by special fixed addresses. For example, on a T414 the input channel of processor link 3 is always at address $8000001C_{16}$. Internal channels have addresses allocated by software.

This equivalence of internal channels to hardware link channels means it is possible to develop a parallel system on a single transputer and then move some of its processes onto other transputers without having to recompile any code.

Each process executing on a transputer processor has a priority, which can either be 'urgent' or 'not urgent'. The processor automatically shares its available time between these processes. A process can be *descheduled* either because it has performed an operation (such as sending a message to another process) which

causes it to pause or, in the case of a 'not urgent' process, because it has been executing without interruption for a certain period of time. The effect of this is that the CPU time-slices between the 'not urgent' processes, but 'urgent' processes are not interrupted until they cannot proceed because of a communication. For this reason, 'urgent' processes should be designed so that they do not perform large amounts of computation, as they will 'lock out' the less urgent processes entirely.

3.3 Software model

Parallel C is based on the same abstract model of communicating sequential processes as the transputer hardware.

A complete *application* is viewed as a collection of one or more concurrently executing *tasks.* Each task has its own region of memory for code and data, a vector of *input ports,* and a vector of *output ports.* The port vectors are passed to the task as arguments to its **`main`** function.

Tasks can be treated as software 'black boxes' connected together via their ports, as shown in figure 3.1.

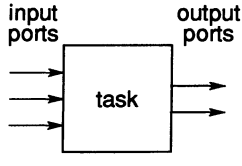


Figure 3.1 a task viewed as a 'black box'.

For example, a very simple task might accept a stream of **`char`** values on an input port, convert each character to upper case, and output the resulting stream of characters on an output port. The C code for this is shown in figure 3.2.

Tasks can be treated as atomic building blocks for parallel systems, to be wired together rather like electronic components. Indeed, several such basic building-block tasks are supplied with the compiler.

Each element in the input and output port vectors is of type 'pointer to channel word', (**`CHAN *`**). Ports are *bound* to real channel addresses by configuration software external to the task itself; the bindings can be changed without recompiling or relinking the task. Extended C run-time library functions supplied with the compiler allow C programs to send and receive messages over the channels bound to a task's ports.

The configuration software also provides ways of specifying which software tasks

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

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

   for (;;) {
      chan_in_word(&c, in_ports[0]);
      if (c == -1) break;   /* terminate task */
      chan_out_word( _toupper(c), out_ports[0] );
   }
}
```

Figure 3.2 Complete example task with one input and one output port.

are to be run on which hardware processors. Each processor can support any number of tasks, limited only by available memory.

Tasks placed on the same processor can have any number of interconnecting channels. Tasks placed on different processors can only be connected where physical wires connect the processors' links. Each logical connection between two tasks placed on different processors is assigned exclusive use of one of the physical link channels connecting the processors. The number of interconnections between tasks on different processors is therefore limited by the number of hardware links each one has. If more than four logical connections in each direction are required between one transputer and its neighbours, the designer of the system must provide explicit multiplexor tasks.

3.4 Parallel execution threads

The model described so far consists of a network of tasks communicating with each other by sending messages over channels. Each task has its own code and data areas.

Parallel C also provides the mechanisms to dynamically create new concurrent threads of execution within a task. Each thread has its own stack, allocated by its creator, but shares its code, static data and heap space with other threads in the same task. Semaphore functions in the run-time library are provided to control access to shared data and channels.

These threads can communicate either by using channels or by using shared data.

Parallel threads must be used if a task must wait for a message on one of a number of input channels. The inputs cannot be performed sequentially as the task would have to wait for input on the first channel regardless of inputs arriving on the other channels.

For example a server task would require a separate thread for each of its clients. Each thread services all requests from one input channel. The thread would consist of a loop which inputs a message and calls a service function for the message.

The parallel thread and semaphore functions provide a powerful extension to C. However incorrect use of these functions can lead to obscure problems in your programs which can be very difficult to find. Unlike the occam parallel constructs, the correct use of these extensions cannot be checked by a compiler.

It is therefore recommended that the use of these functions should be localised within your source code.

3.5 Configuring an application

A multi-transputer program consists of a network of communicating tasks which are distributed over a physical network of transputers.

A configuration file must be created which describes:

- the transputers in the physical network and how they are connected.
- the names of the tasks and how they are connected.
- the placement of the tasks on the transputers.

Multi-transputer programs are built in two stages.

First, each individual task is compiled, linked and converted into an executable image.

Second, the configurer is used to create the final program. The configurer takes as input the configuration file and the task images and creates as output a program which can be loaded to the transputer network with the server.

3.6 Processor farms

The tools described so far allow you to build applications which execute on any transputer network, the wiring of which can be specified in advance in a configuration file. For many parallel computations it is useful to be able to create applications which will automatically configure themselves to run on *any* network of transputers. Such applications will automatically run faster when more transputers are added to a network, without recompilation or reconfiguration.

Parallel C allows you to create applications like this, provided the application can be implemented by a *processor farm,* and provided that there is enough memory on each processor in the network to support the required loading and message handling software.

In the processor farm technique, an application is coded as one *master* task which breaks the job down into small, independent pieces called *work packets* which are processed by any number of anonymous *worker* tasks. Work packets are automatically distributed across an arbitrary network of transputers by *routing* software supplied with the compiler. All of the worker tasks must run the same code. Each worker simply accepts work packets, processes them, and sends back *result packets* via the same routing software. A worker task is just a simple sequential loop: read a packet; process it; send back a result packet.

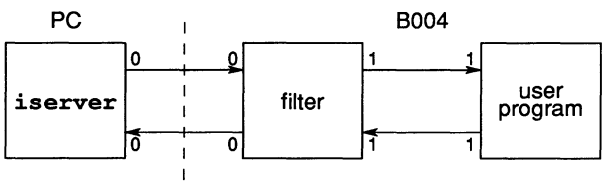
Provided a master task can be written for your application which will split the job up into *independent* work packets which the worker tasks can handle without communicating with other tasks, you can use the *flood-fill configurer* to combine the code for the master and worker tasks into a bootable application file which can be loaded automatically into an arbitrary transputer network by the `iserver` program.

Many computationally intensive applications can in fact be implemented by processor farms, particularly graphics applications like ray-tracing where different sections of the screen can be worked on independently.

4 Programming transputer networks

In this chapter we move on from looking at the general features of Parallel C to explaining how some of the parallel programming tools supplied with the compiler are used in practice. The general-purpose configurer is described here along with the extended run-time library functions for sending messages over channels and creating new execution threads. Processor farm applications are covered in the next chapter.

We have actually already encountered an interesting example of a parallel system: even a simple sequential program running on a transputer board plugged into the host runs in parallel with the **iserver** program on the host computer, as shown below.



The **iserver** task is an application program that runs on the host. It loads executable files onto the transputer and also acts as a file server, handling I/O requests made by the transputer. The **iserver** and the transputer execute in parallel and communicate via an Inmos link. The messages sent to the **iserver** are normally generated by the Parallel C run-time library. It converts I/O operations like **putchar** and **fprintf** into messages requesting the **iserver** to perform host operations like *write 512 bytes* and then waits for the **iserver** to reply.

It is always necessary to plug a task called the 'filter' process between the user process and the **iserver**. The filter runs in parallel with the **iserver** and the user task; it simply passes on messages travelling in both directions. The reason that the 'filter' task is always used is because the protocol generated by the Parallel C run-time library to request host services is different from that used by the **iserver** and so the 'filter' task converts the protocol used by the Parallel C run-time library to that used by the **iserver**

4.1 Configuring one user task

Up to now a standard 'harness', **mainent.c4x** or **mainent.c8x**, has been linked in with all user programs. The harness contains system initialisation code, the filter, and a call to the user program. There is no need to describe the standard system configuration (**iserver**, filter and one user task) to the harness; the configuration is assumed.

The standard harness provides a simple solution for simple cases. We need a way to produce executable files for more complicated system configurations containing many tasks and many transputers. The configurer program is used to build programs for these more complex configurations.

The configurer takes as input a user-written *configuration file* which describes the system to be built: the file lists all the physical processors in the system, the wires connecting them, the tasks to be loaded into the system and their logical interconnections. The complete configuration file needed for a single transputer system with one task (i.e. the same configuration that is built into the standard harness) is shown in figure 4.1. In the rest of this section we will look at its contents in detail.

```
!
! UPPER.CFG
!
processor host        !the host computer
processor root        !the transputer on the
                      !         transputer board
wire  jumper -        !connects...
      root[0] -       !link 0 of root transputer
      host[0]         !to the host computer

task  upper   ins=2 outs=2              !the user task
task  filter  ins=2 outs=2 data=10k
task  iserver ins=1 outs=1

place iserver host  !iserver runs on host computer
place upper root     !everything else on transputer
place filter root

connect ?   filter[0]   iserver[0]
connect ?   iserver[0] filter[0]
connect ?   filter[1]   upper[1]
connect ?   upper[1]    filter[1]
```

Figure 4.1 Configuration file with one example task

The example program we have chosen just converts a stream of characters read from `stdin` to upper case. The C source file, `upper.c` is shown in figure 4.2 (the corresponding configuration file is called `upper.cfg`).

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

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

Figure 4.2 C Source file for upper casing program, `upper.c`

4.1.1 Hardware configuration

The first thing the configuration needs to describe is the hardware configuration. A single transputer board plugged into the host is very easy to describe.

```
processor host
processor root
wire jumper host[0] root[0]
```

There are two processors: the host computer and the root transputer in the transputer board. The root transputer is so called because if a larger network is built around a basic single transputer system, the transputer directly connected to the host computer becomes the root of the network — all communication with the file system on the host computer must pass through it.

A wire connects the root transputer's link 0 to the host computer. The WIRE statement describes actual physical cables, in this case the jumper you have to plug into the transputer board which connects link 0 on the transputer to the host computer. Each wire is given a name, in this case `jumper`. Objects declared in the configuration language can have arbitrary names made up of letters, digits and the special characters '_' and '$', but are usually given mnemonic names.

The processor identifiers (`host` and `root`) used in a WIRE statement must have been declared in a previous PROCESSOR statement. This is a general rule: all objects in the configuration language (processors, wires, tasks) must be declared before they are used.

Now compare the short example with the full configuration file in figure 4.1. You will notice a few differences in layout. Blank lines, spaces and tabs have been used to improve readability, and comments (from a '`!`' character to the end of the

line) have been added. Some lines have been broken, indicated by a hyphen, '–', as the last non-whitespace character before a line break (or comment). Layout and comments are ignored by the configurer. Note that, unlike C, the configurer also ignores the case of letters: 'a' and 'A' are not distinguished.

4.1.2 Software configuration

As well as describing the hardware of a system, the configuration file must contain details of all its software tasks and their interconnections.

Declaring tasks

For each concurrently executing task in the system the configuration file must contain a TASK statement which declares the number of input and output ports the task has. The **iserver** has only one input port (for file system requests) and one output port for responses.

```
task iserver ins=1 outs=1
```

Our example user task is next. It will be a program to convert characters to upper case, so it is given the name **upper**.

```
task upper ins=2 outs=2
```

As before, the **ins** and **outs** attributes specify the number of input and output ports for the task. The **upper** task has two of each, numbered from 0 as in C, and called **upper[0]** and **upper[1]**. Whether a port specifier like **upper[0]** refers to an input or an output port is determined by the context in which it is used.

The ordinary Parallel C run-time library, with which the **upper** task will be linked, makes the assumption that the first two input and output ports of a task will be reserved for its use. The first pair of ports (numbered 0) have uses which will not be described here; they should simply be left unconnected. The second pair of ports (numbered 1) are assumed to be connected to a file server task. Here, we will connect the **upper** task to the **iserver** through a filter task.

The filter task has a slightly more complicated declaration:

```
task filter ins=2 outs=2 data=10k
```

The DATA attribute specifies the amount of memory a task needs. The **filter** task requires a minimum of 10Kb of workspace (used for stack, heap and static data). For ready made tasks supplied with the compiler, like **filter**, memory requirements can be looked up in the data sheets in appendix A.

A user task like **upper** for which no memory requirement is specified gets all the free memory remaining once any other tasks placed on that processor are loaded. 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, this is not allowed. Section 4.7 gives hints on estimating memory requirements in cases where multiple user-written tasks must be placed on the same processor.

Making connections between tasks

The CONNECT statement establishes a channel between two tasks. Because channels (unlike wires) are unidirectional, two CONNECT statements are needed to create channels going in both directions between the **iserver** and the filter.

```
connect ? filter[0] iserver[0]
connect ? iserver[0] filter[0]
```

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. Currently there is no statement which can refer to the identifier declared by a CONNECT statement, so the question marks avoid the bother of naming essentially anonymous connections.

The remaining connections in our example system are written down in the same way:

```
connect ? filter[1] upper[1]
connect ? upper[1] filter[1]
```

Assigning tasks to processors

The placement of tasks on processors is specified by the PLACE statement. In our example, the **iserver** runs on the host computer and the user task (**upper**) runs on the root transputer with the filter task.

```
place iserver host
place upper root
place filter root
```

4.1.3 Building the application

To build a multi-task application we must first build task images for each task (in our example the server and filter tasks have been provided ready built). The

configurer is then used to create the multi-task program image; it takes as input the configuration files and the task images.

Building a task image

The following steps are required to build a task image for the **upper** task.

```
t4c upper
ilink taskharn.t4x upper.bin crtl.lib -o upper.c4x
iboot upper.c4x -c -o upper.b4
```

Notice when building a task harness **taskharn.t4x** is used rather than **mainent.t4x**. Notice also that the **-c** option must be given to the **iboot** program to specify that the output file is for input to the configurer.

Be sure to check that the correct harness is used as the linker is unable to detect this error, and programs created may fail to execute or simply give the wrong answers.

Configuring the task images

To create the program **upper.bt** the configurer is used as follows:

```
config upper.cfg upper.bt
```

The filenames for the task images are derived by appending **.b4** to the task names in the configuration file. If the task images are not found in the current directory the search path is used (so **filter.b4** will be found in the 'standard' libraries directory).

Note that tasks running on the host transputer are not searched for. These are named for the notational convenience of describing connections to the host.

The program file output by the configurer can now be run using the server:

```
iserver -sb upper.bt
```

The actual configuration of the network attached to your computer must of course match the description in the configuration file.

4.2 More than one user task

In the previous section we saw how an application consisting of a single user task could be built using the configurer instead of the standard harness.

From this base, we can move on to more complicated systems containing multiple user tasks running in parallel.

Let's continue with the small case conversion example by splitting the job performed by `upper.c` into two tasks: 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.

4.2.1 Inter-task communication functions

Coding the driver task in C is easy. Instead of using the `_toupper` macro 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 described in chapter 15. The `chan` package provides functions to send and receive messages of any length. The driver task is shown in figure 4.3; it uses `chan_in_word` and `chan_out_word` to handle word-sized messages. A word is the same size as an `int`, 32 bits.

The driver source file, `driver.c`, is included as an example in the distribution kit, along with the processing task, `upc.c`, and a suitable configuration file, `upc.cfg`. These files can be found in the `examples` subdirectory of the directory containing the compiler.

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

```
chan_out_word( c, out_ports[2] );
```

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

Beware when using the channel I/O functions that sending and receiving tasks always agree on the size of messages. For example, if a task sends a word value as a single 4-byte message, the receiving task *must* read it as one 4-byte unit; it is not possible for the receiving task to read four separate 1-byte messages. Trying to do so may cause the transputer to lock up or behave unpredictably.

The second argument to `chan_out_word` identifies the output port to which

```
/* driver.c file I/O for uppercasing example */

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

main(argc, argv, envp, in_ports, ins, out_ports, outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_ports[], *out_ports[];
{
   int c;
   for (;;) {
      c = getchar();
      chan_out_word( c, out_ports[2] );
      if (c == EOF) break;
      chan_in_word( &c, in_ports[2] );
      putchar(c);
   }
}
```

Figure 4.3 **driver.c** with Channel I/O Calls

the message is to be sent. **out_ports[2]** corresponds to output port 2 of the driver task. A CONNECT statement in the application's configuration file referring to **driver[2]** will specify which task the port is connected to. In our case, it will be the processing task to be described later.

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 is included by C files which use the channel I/O functions. Each port (i.e. each element in the vector) has type 'pointer to channel'.

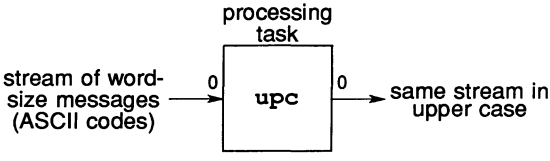
The number of output ports in the vector is defined by the OUTS attribute of the TASK statement used to declare the task in the configuration file. Our **driver** task has **ins**=3, so there are three elements in its output port vector, numbered 0 to 2.

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 which 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:



A Parallel C implementation of this task is shown in figure 4.4.

```
/* upc.c standalone processing task; */
/*                communicates with driver.c */

#include <chan.h>
#include <ctype.h>

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

   for (;;) {
      chan_in_word(&c, in_ports[0]);
      if (c == -1) break;    /* terminate task */
      chan_out_word( _toupper(c), out_ports[0] );
   }
}
```

Figure 4.4 The Processing Task

The processing task uses the same channel I/O functions as the driver to send and receive messages. It terminates when it receives a −1 from the driver. (The character codes are sent as words rather than bytes because in this implementation of C, **char** variables can only hold values in the range 0 to 255; −1 is not a valid **char** value).

Extending the configuration file for our first, single-task, example (fig. 4.1) to handle two tasks is easy. We just change references to the old **upper** task to **driver**, and add the following extra configuration statements to describe the processing task and its connections.

```
task upc ins=1 outs=1 data=5k
place upc root
connect ? driver[2] upc[0]
connect ? upc[0] driver[2]
```

This says that the new task **upc** has one input port, one output port, and requires 5KB of memory (section 4.7 gives hints on estimating task memory requirements). The **upc** task is placed on the root transputer, and its ports are connected to the corresponding ports of the driver task.

4.2.2 Building the application

As with the previous example the program is built by first creating task images for each task in the system, and then using the configurer to create the multi-task program.

```
t4c driver
ilink taskharn.t4x driver.bin crtl.lib -o driver.c4x
iboot driver.c4x -c -o driver.b4
```

The sequence above creates the task image driver **driver.b4**.

```
t4c upc
ilink taskharn.t4x upc.bin sacrtl.lib -o upc.c4x
iboot upc.c4x -c -o upc.b4
```

The sequence above creates the task image **upc.b4**. It is important to notice that the reduced C run-time library **sacrtl.lib** is used rather that **crtl.lib**. This is because the process **upc** does not perform any host input or output. A full explanation of the use of this library is given in the next section.

```
config upc.cfg upc.bt
```

creates the multi-task program **upc.bt** which can run using the server as follows:

```
iserver -sb upc.bt
xyz123
XYZ123
pqr
PQR
'EOF'
```

You should try this out for yourself using the example sources provided.

4.3 Access to host services

This section explains how programs running on transputers access host services such as file and terminal handling. It is important that you understand this if you are building multi-task programs.

The basic principle of operation is quite simple. Requests are passed to an output port (through the **filter** process) to the server, the server processes the request and returns the result to the task on an input port (via the **filter** process).

As ports are point to point connections only one task, the *iotask*, may be connected to the server in this way. An *iotask* is a task that performs i/o operations with the host file system, for example, using the standard i/o functions **getchar**, **printf** etc. Only the *iotask* may use functions such as **printf** which require access to the host machine. All other tasks 'compute tasks' can only perform input and output through ports.

The *iotask* is linked with the full run-time library **crtl.lib**. The library makes input and output requests on output port 1 and receives replies on input port 1. The task must not use these ports for direct port input and output.

All compute tasks *must* be linked with the *reduced* run-time library **sacrtl.lib**, otherwise the program will fail to operate correctly.

4.4 Multi-transputer systems

If you have followed the examples this far, the generalisation from a multi-task system running on a single transputer to a full multi-transputer 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 transputer.

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

```
processor addon
wire ? root[1] addon[0]
```

This gives a name (**addon**) to the second processor and declares that it will be connected by a wire from its link 0 to link 1 on the root transputer. (Link 0 on the root transputer is already being used to connect it to the host computer).

If we reconfigured the application at this stage, the **addon** processor would

be unused because the **upc** and **driver** tasks are both placed on the root transputer. We can fix this by modifying the PLACE statement for **upc**.

```
place upc addon
```

Now the configurer will automatically generate all the bootstrap and loader software required to make sure that the code of the **upc** task is loaded into the second transputer when the complete application is started on the root transputer by the **iserver**.

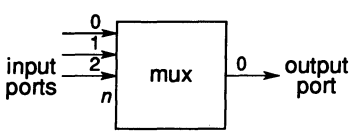
```
config upc.cfg upc.bt

iserver -sb upc.bt
two transputers...
TWO TRANSPUTERS...
'EOF'
```

Further generalisation to an arbitrary system should be clear: just declare more processors and wires in the configuration file, place tasks on the processors and connect them together.

4.5 Multi-threaded tasks

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 which 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. It might be used to allow a number of tasks to share a single hardware link.



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.

4.5.1 Creating threads

To implement the *mux* task we will need a way of reading from a number of input ports 'all at the same time' so that the first message to appear on any of them 'wins' and satisfies the read request, blocking any other messages which appear until the next read request.

In Parallel C this can be done by creating a new *execution thread* for each input port. Each thread in our example does a simple sequential read and waits for a message. As soon as a thread receives a message it waits until a *semaphore* indicates the output port is free. It needs to wait in case one of the other threads is currently using it. Using a semaphore prevents disaster if two threads each try to write to a shared object like the output port at the same time.

Figure 4.5 shows an implementation of the multiplexor task in Parallel C. 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_create`** function. Any number of input threads can 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.

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 [6,5], or restrict yourself to the stylised usage shown in the example. It is possible to introduce difficult-to-trace errors into a program if threads forget to synchronize access to a shared object by waiting for a semaphore.

```
/* mux.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 */
main(argc, argv, envp, in_ports, ins, out_ports, outs)
int argc, ins, outs;
char *argv[], *envp[];
CHAN *in_ports[], *out_ports[];
{
   extern void receive();
   int i;
   sema_init( &buf_free, 1 );/*buffer initially free*/

   in_p = in_ports;     /* make in_ports & out_ports */
   out_p = out_ports;   /* globally available */

   for (i=0; i < ins; i++) /*1 thread per input port*/
      thread_create( receive,           /* function */
                     50*sizeof(int),
                         /* workspace size in bytes */
                     1,                /* 1 argument */
                     i ); /* tell thread which port */
}
void receive(i)            /* handle a single input port */
int i;                     /* which input port to service */
{
   int msglen;     /* each thread has its own msglen */
   for (;;) {      /* forever... */
      chan_in_word(&msglen, in_p[i]);
                   /* await message from input port */
      sema_wait(&buf_free);
                /* wait till no one else using buf */
         chan_in_message( msglen, &buf[0], in_p[i] );
            /* read body of message into the shared */
            /* global buffer from our port */
         chan_out_word(msglen, out_p[0]);
                    /* copy message to out_ports[0] */
         chan_out_message(msglen, &buf[0], out_p[0]);
      sema_signal(&buf_free);
                     /* let someone else in again */
   }
}
```

Figure 4.5 Multiplexor Task Using Semaphores

4.5.2 Threads versus tasks

Threads can be useful in many situations. They are just 'lightweight' processes, corresponding to processes in Modula-2 or the co-routines of some other languages.

Compared with tasks, threads are:

- 'lightweight'—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 by using channels like tasks, or via shared memory;
- 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 than threads:

- 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; this is so that
- 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 relinking.
- 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 it is modified.

4.6 Debugging

What can be done when a parallel system locks up or fails to work properly? A sequential program could be attacked by inserting extra debugging output statements at strategic points in the code.

In a multi-task system this will in general only be easy to do to an I/O server task linked with the standard library and directly connected to the **iserver**. Unless you design debugging messages into the communication protocol used between the various tasks in your system you will not be able to get debugging output from a standalone task to a screen driving task. Even building debug message formats into the protocols used by the tasks in your system may not be enough if the fault lies in the failure of some intermediate task to transmit messages correctly.

However, it is possible to get output directly from a standalone task to an output device by using a second host computer and transputer board combination as a debugging tool. The second system can be attached to a suspect node of the system, in the same way as an oscilloscope can be used to debug an electronic system.

One way of doing this is to relink the suspect task with the standard run-time library (rather than the standalone library) and place it on the transputer attached to the second host computer. Ordinary **printf** calls can then be inserted in the code; the results will be output directly by the **iserver** in the second host computer and displayed on its screen. The configuration statements required would be like this:

```
processor host
processor root
wire ? root[0] host[0]          !as before
processor extra_host type=PC
processor extra_processor  !plugged into
                           ! extra_host
task extra_iserver ins=1 outs=1
wire ? extra_processor[0] extra_host[0]
wire ? extra_processor[1] root[1]

place extra_iserver extra_host
place suspect_task extra_processor

connect ? suspect_task[1] extra_iserver[0]
connect ? extra_iserver[0] suspect_task[1]
```

The main thing to notice here is the **type=PC** attribute given to the **extra_host** processor. This tells the configurer not to try and bootstrap any tasks into that

processor. (The **host** processor is just a special case for which **type=PC** is assumed). To make this configuration work, you must start the **iserver** on the extra host computer using the **iserver** command with just the the **SS** option before starting the system under test. If just the **SS** option is present on the command line, the **iserver** does not attempt to bootstrap the network it is attached to; it will simply accept file I/O request messages over its links.

It is also possible to use this debugging technique if you don't have another host and transputer board combination but do have another host computer with an INMOS link adapter card. Relink the suspect task with the full run-time library rather than the standalone library, then reconfigure the system with input and output ports 1 of the task being debugged connected to the host computer with the link adapter, as follows:

```
processor second_host type=pc
task second_iserver ins=1 outs=2
place second_iserver second_host

processor any_processor
                          !of network being debugged
wire ? any_processor[3] second_host[0]

task suspect_task ins=2 outs=2
                           !connect [1]'s to iserver
place suspect_task any_processor
connect ? suspect_task[1] second_iserver[0]
connect ? second_iserver[0] suspect_task[1]
```

This technique has two advantages: it only requires an extra host computer and link adapter card, rather than an extra host computer and transputer board, and there is no need to change the placement of the suspect task.

A third technique uses the three spare links on a transputer board plugged into the extra host computer to accept debugging messages from up to four separate tasks anywhere in the network being debugged and multiplex them onto the host computer's screen.

4.7 Estimating memory requirements

The data requirement for a task is the sum of the number of bytes required for stack (**auto**), **static**, **extern** and heap storage in all its modules.

The **decode** utility (see chapter 7) can be used to determine a module's static data requirement (including **extern** data). **decode** displays the number of *words* (not bytes) of static data required by a module near the top of the output

listing it produces, after the keyword STATIC. The whole task also has one word of static space permanently allocated to each module.

Stack and heap requirements are more difficult to estimate; you must decide how much space to leave for all the functions which may be active at once, based on the sizes of individual data items. Each level of function calling uses about five words of stack space in addition to the space required for function data.

Heap storage is currently allocated by the run-time library in blocks of 4KB, 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 it is a good idea to leave at least 1 or 2KB of extra overflow space, unless you are absolutely sure the task will never require more space than you have calculated.

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. A good rule of thumb would be to allocate at least 1KB to simple tasks which don't use the heap, and 8–10KB for tasks which do use the heap. Note that the C standard I/O functions implicitly use the heap to allocate buffer space.

If the stack space required by a task is small enough it can be allocated from the transputer's on-chip RAM. The space available there is 2KB on a T414, 4KB on a T800. Placing a computationally intensive task's stack in fast on-chip RAM can produce dramatic speed improvements. The configuration language contains various attributes for the TASK statement which allow control over memory layout. These more advanced topics are covered in chapter 17.

5 Processor farms

The previous chapter showed how to create a parallel application for a multi-transputer system with a fixed hardware configuration. In this chapter we look at how to build one of the 'processor farm' applications mentioned in the *Introduction to Parallel C* in chapter 3 which will automatically flood-fill an arbitrary network of transputers with copies of a 'worker' task.

Three things must be written to create a processor farm application:

1 A master task to split up the job into independent work packets.

2 A worker task, which is automatically copied to each node of the network.

3 A configuration file, describing the memory requirements and other attributes of the tasks.

In this chapter we will use a program which multiplies two matrices together as an example processor farm application.

The full source code of the matrix multiplication master and worker tasks, and of the configuration file required, is supplied in machine-readable form in the **`examples`** subdirectory in the release directory, along with a command file (**`matrix`**) which compiles, links and configures the example files into an executable application. Section 5.4.4 at the end of this chapter explains how to run the demonstration if you want to try it out before reading further.

The matrix multiplication program is suitable for running on a processor farm because each element of the final matrix can be computed independently of all the others.

The master task has to split the job up into lots of small units which can be handled independently by the 'farm workers'. In the matrix multiplication case this is easy: the master divides up the calculation of the output matrix and sends the rows and columns of the input matrices to be multiplied out into the network as work packets. Any idle worker receiving a data packet calculates the required result and sends it back as a result packet.

5.1 The worker task

If you look at the code of the matrix multiplication worker task you will see that it is purely sequential. It consists of a single loop:

1 Get the work packets by calling **net_receive**. These work packets identify the element in the output matrix to be calculated and the row and column data of the input matrices to be used for this calculation.

2 Work out the value of the element in the output matrix using the row and column data of the input matrices.

3 Send the result packet back to the master task by calling **net_send**.

4 Go back to step 1.

The **net_send** and **net_receive** functions are described below in section 5.3.

The worker task does not care which processor it is executed on and must not communicate explicitly with other tasks. All communication between workers and master is handled 'behind the scenes' by **net_send** and **net_receive**.

The only other restriction on the worker task is that because it must be replicated throughout the network and therefore cannot be directly connected to the **iserver** it must be linked with the reduced run-time library (see section 4.3).

5.2 The master task

The master task of a processor farm application has three basic functions.

1 Split up the job into work packets. It sends the work packets out into the farm of worker tasks by calling **net_send**. The master simply does this as fast as it can: whenever the network of worker tasks becomes saturated, **net_send** is automatically blocked until a worker task becomes idle.

2 Receive result packets from the network by calling **net_receive**. If no result packets are available, **net_receive** will wait for one to arrive before returning.

3 Perform any I/O required for writing out the results received from the worker tasks.

To prevent incoming result packets being blocked by the **net_send** function waiting for a worker to become free, or conversely the sending of work packets being blocked by **net_receive** waiting for a reply, these functions must be performed in parallel.

In the example implementation of the matrix multiplication program these functions are performed by two parallel execution threads: **send** and **main**, which

are synchronised using semaphores.

5.3 The net functions

The **net_send** and **net_receive** functions used by the master and worker tasks must be declared by including the appropriate header file:

```
#include <net.h>
```

The **net** functions provide a procedural interface to the underlying message-based software which routes work packets from the master to free worker tasks and carries result packets back again.

net_send has three arguments:

```
int net_send(nbytes, packet, complete)
char *packet;
int nbytes, complete;
```

If **net_send** is called by the master task, the message packet is sent to any free worker task; if the function is called by a worker task, the packet is sent back to the master task.

nbytes is the number of bytes of data in the buffer pointed to by **packet**.

If **nbytes** is less than zero or greater than **NET_MAX_PACKET_LENGTH** (defined in version 2.0 of Parallel C by **<net.h>** to be 1024 bytes) no message is sent and the function returns a negative value.

Otherwise the function returns the number of bytes sent, which will be **nbytes** if no error occurs.

If a message longer than **NET_MAX_PACKET_LENGTH** has to be sent, it must be broken up into a number of packets, each smaller than this limit.

If **complete** is 0, the argument **packet** is regarded as part of a larger message; a circuit to the destination task is held open until the last packet of the message has been sent. The final (or only) packet of a message is marked by setting **complete** equal to 1.

The routing software guarantees that multiple packets sent in this way are always received by the destination task in the same order they were sent.

In normal use, packets will be smaller than 1024 bytes and **complete**

will always be given the value 1. Sending very long packets can clog up the network, blocking packets being delivered to other nodes.

`net_receive` has two arguments:

```
int net_receive(packet, complete)
char *packet;
int *complete;
```

The next (or only) packet of the message being received is read into the buffer pointed to by `packet`.

If `net_receive` is called by the master task it reads the next available result packet returned by a worker task; if it is called from a worker task, it reads the next work packet sent out by the master.

The size of the packet (in bytes) is returned as the result of the function.

If the `packet` is the final or only packet of the message, the location pointed to by `complete` will be set to 1; otherwise it is set to 0 and the receiving task must repeatedly call `net_receive` to read the remaining part of the message.

No more than `NET_MAX_PACKET_LENGTH` bytes will be read into the `packet` buffer. Less space may be allocated if it is certain that the sending task will not send messages longer than some smaller limit (for example, if only fixed-length messages are being used).

5.4 Building the application

5.4.1 Building master and worker task images

The following sequences of commands are used to build the master and worker file images:

```
t4c matrixm
ilink taskharn.t4x matrixm.bin crtl.lib -o matrixm.c4x
iboot matrixm.c4x -c -o matrixm.b4

t4c matrixw
ilink taskharn.t4x matrixw.bin sacrtl.lib -o matrixw.c4x
iboot matrixw.c4x -c -o matrixw.b4
```

Note that the worker task is linked with the reduced run-time library `sacrtl.lib` (see section 4.3).

5.4.2 Configuration file

Like the fixed-network configurer, **fconfig** requires a configuration file as input. This must specify at least:

- the filename of the master task;
- the filename of the worker task;
- the memory requirements of the worker task.

The configuration language accepted by **fconfig** is a subset of that accepted by **config**.

The minimum configuration file for the matrix multiplication example would be:

```
task master
task worker data=10k
```

fconfig would search for the master task in **master.b4**, and for the worker task in **worker.b4**. These file names can be over-ridden using the FILE attribute of the TASK statement, as shown below, but the task identifiers **master** and **worker** are special: you must use these names to identify the master and worker tasks to the flood-configurer.

If the alternative configuration file below was used, the configurer would expect to find the tasks in files called **matrixm.b4** and **matrixw.b4**.

```
task master file=matrixm
task worker file=matrixw data=10k
```

The DATA size specification is required for at least one of the tasks. Other attributes governing placement of stack memory in on-chip RAM and so on are covered in the reference part of this manual (see chapter 17).

INS and OUTS must not be specified for the master and worker tasks. All ports and connections are handled automatically by the configurer.

5.4.3 Configuration

The flood-fill configurer is invoked to create the matrix multiplication program as follows:

```
fconfig fmatrix.cfg fmatrix.bt
```

The executable file generated by the flood-configurer will place the master task

and one copy of the worker task on the root transputer, and distribute copies of the worker task to any other transputers connected to the root. A filter task allowing the master task to communicate with the **iserver** is automatically added by **fconfig**, along with the loader and router tasks required to copy the workers across the network and carry messages between them and the master task.

This additional software occupies about 20KB of RAM in version 2.0 of Parallel C, so each node in our example network must have at least 32KB of RAM to support the 10KB worker task declared in the configuration file along with a router and loader. The root node must be larger again in order to support the master task as well.

5.4.4 Running the example

A command file, **matrix**, is supplied, along with the matrix multiplication example program source files, which will automatically compile, link and configure the application. These files can be found in the **examples** subdirectory in the release directory.

The resulting executable file (called **fmatrix.bt**) can be loaded and run on any network containing only T414 transputers. To use T800 transputers you would have to recompile the tasks to generate T800 code. Section 5.5 below describes how to flood-configure applications to run on a network containing a mixture of T414 and T800 processors.

The executable file can be loaded and run in the normal way:

iserver -sb fmatrix.bt m_1 m_2 ... m_n *output*

where: m_1 m_2 ... m_n are the input matrices

output is the output matrix filename.

Example matrices are supplied along with the source for the program. The format of the output file is identical to that of the input file. As an example, try the following, using the supplied example matrix files **mat1.m33** and **mat2.m32**:

```
iserver -sb fmatrix.bt mat1.m33 mat2.m32 mat.m32
```

Once you have the program working, you can make it run faster simply by plugging more T414 (or T800) transputers into the network.

5.5 Mixed networks

A flood-filled application compiled for the T414 and configured using the simple **master** and **worker** forms of task declaration may work on a mixed network of T414 and T800 processors if it uses only integer operations. This approach will not in general work for an application which uses floating-point operations, because of the incompatibilities between the T414 and T800 instruction sets.

Mixed networks of T414 and T800 processors are properly handled by an extension to the configuration file, like this:

```
task t4master file=matrixm4
task t8master file=matrixm8
task t4worker file=matrixw4 data=10k
task t8worker file=matrixw8 data=10k opt=stack
```

Separate tasks must be compiled and linked for T414 and T800 processors; the Parallel C software ensures that the right task images are loaded into the right processors.

Again the names **t4master**, **t8master**, **t4worker** and **t8worker** are special, but the file names derived from them can be over-ridden by the FILE attribute, as above.

Note that it is possible to specify different memory optimisation options (e.g. **opt=stack** above) for the T414 and T800 variants of a task. This is useful because the T414 and T800 have different amounts of on-chip RAM.

If a **t4master** task is declared, a corresponding **t8master** task must also be declared, and similarly for the worker task.

The reference manual

72 TDS 179 00

6 `config` general purpose configurer

The general purpose configurer is used to build multi-task multi-transputer programs. A task image must be created for each task by using the `iboot` tool. A configuration file must be created to describe the network, the tasks and interconnecting processors, and how the tasks are distributed over the processor network.

The configurer takes as input the configuration file and the task images and creates a program which can be automatically loaded to the processor using the server.

6.1 Running the configurer

The configurer has a simple command line:

`config` *configuration-file output-file*

All options are defined in the configuration file. These are discussed in the chapter describing the configuration language (chapter 17).

The configuration file also defines the task image files required by the configurer. If these are not found in the current directory the configurer will use the search path.

6.2 The distributing loader

This chapter provides a summary of the way in which a single-transputer system is bootstrapped, and a description of the way in which this method is extended to handle multi-transputer systems. The distributing loader program is described and the protocol which it uses is covered in detail.

6.2.1 Bootstrapping a transputer

The transputer processors [10,11] have been designed to function with the minimum of external components. One of the consequences of this design decision has been that the transputer processor contains firmware to make bootstrapping transputer systems possible without supplying each transputer with a bootstrap ROM. A transputer may be configured (using a pin called **BootFromRom**) to bootstrap itself either from an external ROM memory device or from its INMOS

links.

Most transputer development hardware uses the *boot from link* option. Others, such as the INMOS IMS B002 — which contains diagnostic routines and a loader in EPROM — provide a simple switch to disable the bootstrap ROM.

When a hardware reset is performed on a transputer configured to boot from its links, it enters a special state in which messages received on its links can be used to *peek* and *poke* (i.e. read and write) its memory. This state and the link messages used are described in the transputer processor data sheets [10,11] and in the *'Transputer instruction set: a compiler writer's guide'* [12]. The state in which peek and poke functions are available is left when a bootstrap message is received along any link; this message contains the initial program to be executed by the transputer (the *primary bootstrap*). The size of the primary bootstrap is limited by the fact that the bootstrap message contains the size of the primary bootstrap program encoded in a byte value (i.e. the primary bootstrap cannot exceed 255 bytes).

The primary bootstrap is provided with information about the transputer state by the link bootstrap firmware which loaded it. In particular, the primary bootstrap is provided with a pointer to the link from which it was loaded, so that it can use that link to load further information.

It would be possible in principle for the primary bootstrap to load the program to be executed directly from the link at this point, but it has become conventional to interpose a program known as the *secondary bootstrap*. This is invoked by the primary bootstrap and allows the actual user program image to be reasonably configuration-independent; for example, the secondary bootstrap deduces the amount of physical memory available. The bootstrapping scheme used by the 3L language compilers uses this two-stage system.

The standard **iboot** program as provided with the INMOS sequential language compilers produces an output program image file (conventionally given the extension **.b4x** or **.b8x** depending on whether it is for a T414 or T800) which can be used to load a single transputer. The bootstrap process performed by the **iserver** program is simply to reset the transputer and then to write the **.b4x** (or **.b8x**) file to the link one byte at a time. The structure of this file is shown in figure 6.1.

The **.b4x** (or **.b8x**) file for a single-transputer system consists of:

- The primary bootstrap ('PRI' in the figure). As described above, this is a short sequence of code preceeded by a byte containing the length of the bootstrap; the code is loaded into the transputer's memory by the transputer's firmware. The main function of the primary bootstrap is to load the secondary bootstrap into the transputer's memory, but it also

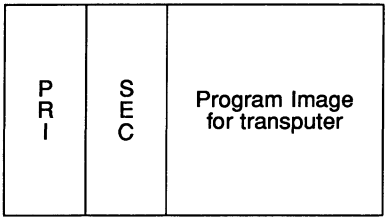


Figure 6.1 **.b4x** (or **.b8x**) File for Single-transputer System

has the task of initialising the transputer's link channel words and other critical processor registers such as the process scheduling queues and timers.

- The secondary bootstrap ('SEC' in the figure). This code sequence is preceded by a 32-bit word containing the length of the code of the secondary bootstrap. The primary bootstrap loads the secondary bootstrap by reading (from the link down which the primary bootstrap was loaded) first the length word and then the code block. The secondary bootstrap has the task of determining how much external memory is available on the transputer and then loading the code of the program to be executed.

- The code of the program to be executed in the transputer. This is preceeded by a number of parameter words which are used by the secondary bootstrap:

  - Target processor type
  - File format version number
  - Harness workspace requirement
  - Harness additional workspace requirement
  - Parallel language stack size
  - Program entry point offset
  - Program code size

After loading the program, the secondary bootstrap executes it, providing to the program such parameters as the size and location of the area of memory which it should use as a workspace, the links to communicate on, and so on. At the same time, having sent the last byte of the **.b4x** (or **.b8x**) file to the transputer's link, the **iserver** program begins to wait for the first filer request from the newly loaded program.

6.2.2 Bootstrapping a network

It would be possible to bootstrap a network of transputers very easily if the host were to posess one link for each transputer in the network; each transputer could then be bootstrapped independently of the others using the method described in section 6.2.1.

This approach is not used in practice because of the large number of links which the host would have to have into the network. Instead, networks are loaded by taking advantage of the observation that any useful network of transputers will be *connected*, i.e. that there is some *path* through the network available between any two processors in the network. If this were not so, some processors in the network would be isolated from the rest of the network and unable to communicate with it.

Because the network to be loaded is connected, it suffices to have only one transputer in the network physically connected to the host. This is referred to as the *root* transputer of the network. After the root transputer has been bootstrapped with a suitable *loader* program, it can be used to bootstrap its neighbours in the network, this process proceeding until every processor in the network has been loaded.

To allow the whole network to be bootstrapped through the root transputer, a more complex executable file must be produced. The standard INMOS tool for making programs executable, **iboot**, is not at present suited to this task and the executable file must be produced by the *configurer* program described elsewhere in this document. The **.bt** file produced by the 3L configurer is very similar to that created for the single-transputer scheme described above. It is shown in figure 6.2.

Note that the **iserver** program has no way of differentiating between the multi-transputer kind of **.bt** file and the simpler kind destined for a single transputer. The root transputer's link is still driven by the **iserver** sending each byte of the **.bt** file in sequence, and when the **iserver** has finished sending the **.bt** file it still enters a loop to service file server messages.

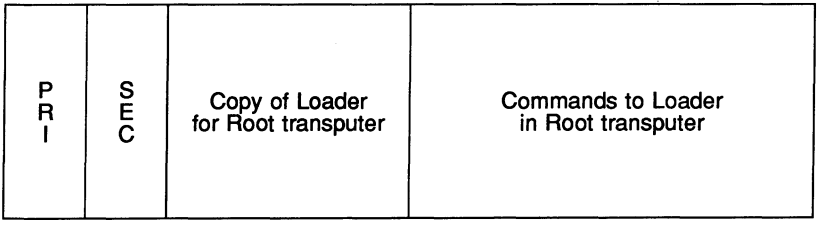


Figure 6.2 **.bt** File for Multi-transputer System

The multi-transputer `.bt` file starts with the same sequence of sections as in the single-transputer case: primary bootstrap, secondary bootstrap, program image. For the multi-transputer case, however, the program image is not a user program but a loader program. In addition, the loader program image is followed in the `.bt` file by a stream of data which will follow the loader program into the root transputer and will be used by the loader as a sequence of commands.

6.2.3 Loader command stream

The command stream for the loader is divided into a series of command *packets*, each of which takes the form shown in figure 6.3.

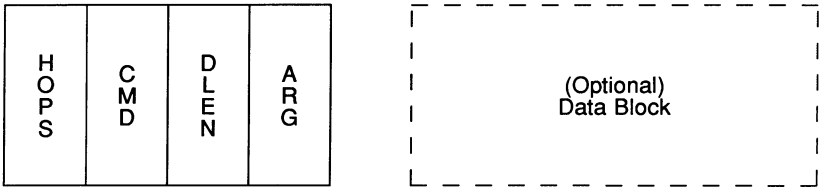


Figure 6.3 Structure of a Loader Command Packet

Each command packet starts with a fixed-length message containing four 32-bit words:

- HOPS contains the distance from the root transputer to the transputer for which this command packet is destined. For example, packets destined for the root transputer would have HOPS = 0, while packets destined for transputers directly connected to the root would have HOPS = 1.

  If a transputer passes a packet on to one of its neighbours, it first decrements the HOPS field. Thus, in a packet in transit within the network, HOPS always contains the number of transputers a packet has still to pass through before it reaches its destination.

- CMD is the command to be executed.

- DLEN is the length of the optional data block.

- ARG is a single integer parameter to the command, used by commands which require only a simple parameter of this type.

If DLEN is zero, the packet header just described constitutes the whole of the packet. If DLEN is not zero, a data block of length DLEN bytes[1] follows. This

[1] To avoid ever transmitting a message of length 1 byte along a hard link, an actual data length of 2 bytes is used should DLEN=1. This is intended to avoid a bug in the link hardware of the Rev A T414 processor.

data block is an additional argument to the command; most information being distributed is carried in the data block.

The data block is limited to 1024 bytes in order to limit the size of buffer which each loader program in the network needs to reserve.

The available command values are as follows:

OPEN Each loader holds a bit-mask of the physical links available to it, with the least-significant bit of the mask representing link 0. Initially this bit-mask is zero, but it is overwritten by the ARG field of an OPEN command. The bit-mask is used under the following two circumstances:

- When a command arrives with HOPS $\neq 0$, the command is sent in turn to all links whose bit is set in the bit-mask. As described earlier, the HOPS field of a packet passed on in this fashion is decremented before transmission.
- By the BOOT command, described below.

There is no data block associated with this command.

BOOT This command is used to bootstrap neighbouring processors. The data block accompanying the command is sent in turn to each link whose bit is set in the link bit-mask set by the last OPEN command. The ARG field is not used with this command.

TASK Creates a new task on this processor. The ARG field contains a bit-mask of flags of properties of the task, as follows:

- PRIO (value 0001_{16}) contains the priority at which the task's initial thread will be started.
- **STACK_DIR** (value 0002_{16}) determines the way in which the task's stack area will be allocated. If this bit is 0, the stack area is allocated from the more positively addressed end of the transputer's unallocated memory, that is at the end farthest from the on-chip RAM. If this bit is 1, the stack area will be allocated from the on-chip RAM end of the transputer's unallocated memory.
- **CODE_DIR** (value 0004_{16}) determines the way in which the task's code area is allocated along the same scheme as described above for **STACK_DIR**.
- **STATIC_DIR** (value 0008_{16}) controls the allocation of the task's static/heap area as described for **CODE_DIR** and **STACK_DIR**.

- **SEPARATE** (value 0010_{16}) controls whether the task has separate stack and static/heap areas. If **SEPARATE** = 1, separate areas are provided. If **SEPARATE** = 0, the two areas are combined.

The data block accompanying the TASK command contains additional information about the task, shown in figure 6.4.

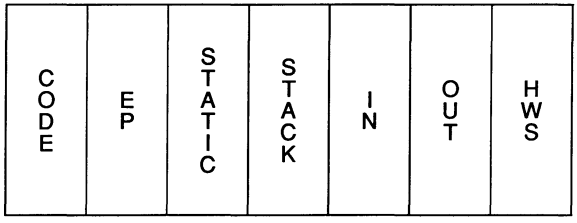


Figure 6.4 TASK Command's Data Block

Each entry in the TASK command's data block is a 32-bit integer, with the following interpretations:

CODE Size of the code area, in bytes.

EP Offset from the start of the code area to the first instruction in the task.

STATIC Size of the combined static and heap area for the task.

STACK Size of the stack for the task.

IN Number of ports in the task's input vector.

OUT Number of ports in the task's output vector.

HWS Size of the workspace required to start the task, i.e. the size of the workspace used by the task's harness.

The various memory areas to be associated with the task are allocated; an internal pointer is set to the start of the task's code area for use by subsequent LOAD commands as described below. If a memory allocation of −1 is specified for STATIC or STACK, the allocation is deferred until the task is started, at which point all available memory is allocated to the appropriate area. Only one such allocation should occur on any processor.

The order of allocation of the various memory areas for the task is performed in a fixed order: stack, code and then static. This order has been chosen so that, in conjunction with the allocation direction bits of the task flags, the most important areas of selected tasks can be placed into the

transputer's on-chip RAM.

If the stack and static areas are to be combined (as specified by the SEPARATE flag described above being set to 0) then the allocation for the combined area is placed in STACK. Memory allocated by STATIC is lost, and the contents of STATIC would therefore normally be 0 under these circumstances.

LOAD The data block accompanying this command contains data to be loaded at the current load point, as initialised by the most recent TASK command. Afterwards, the current load point is stepped over the data just loaded.

CHANS The ARG field of this command is the number of 'extra' channels required for this processor. Such channels are those connected to one or more tasks which are not accounted for by channels associated with physical links. For example, one such channel is required for every connection leading from one task to another task placed on the same processor. No data block is associated with this command.

The loader program allocates a vector of channel words of the requested size, and initialises each element of the array to `NotProcess_P` (the value of `MOSTNEG INT`, 80000000_{16} for 32 bit transputers), thus indicating that no process is currently communicating on the channel.

B_IN ARG must be the number of input ports of the task created by the most recent TASK command, and the accompanying data block contains two 32-bit words for each port. The first of the two words for a particular port contains a value to be used for the port; the second is a description of the kind of value this is. The second word must either have the value 0 (indicating that the value is to be left unchanged) or 1 (indicating that the value should be replaced with a pointer to the 'extra' channel of which it is the index).

B_OUT As for B_IN, but the output ports are bound rather than the input ports.

GO This command is the last one which any loader program receives. It causes the loader to start each task which has been created on the transputer, and to stop itself.

6.2.4 Memory allocation

The distributing loader is responsible for the allocation of physical memory to the various tasks loaded onto a processor. This memory is used for the stack, heap, and code areas of a task, for the 'extra' channels allocated by the loader and for various data structures used by the loader itself.

As an example, take a T414 transputer with 2MB of external memory. The memory map of this configuration is shown in figure 6.5a. The fast 'on-chip' RAM which is built into the transputer processor appears from the lowest memory address and extends for 2048 bytes, occupying addresses 80000000_{16} to $800007FF_{16}$. The slower external memory takes over from the on-chip RAM at address 80000800_{16} and continues to the highest valid memory address in the configuration, which in this case is $801FFFFF_{16}$.

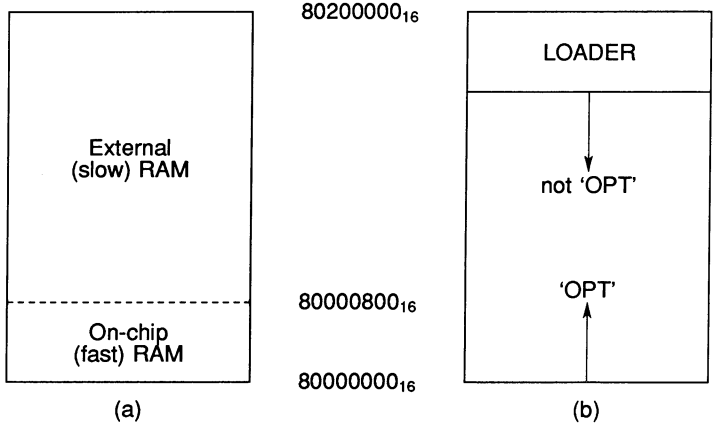


Figure 6.5 Memory Allocation in a 2MB, T414 System

For a T800 transputer processor, the memory map shown would be unchanged except for the upper limit of on-chip RAM, which would be at address $80000FFF_{16}$.

The amount of external memory actually available to a particular transputer processor is established by the secondary bootstrap. The scheme used by the distributing loader's secondary bootstrap establishes the amount of available external memory in multiples of 64KB. Thus, the smallest amount of memory which is acceptable with this scheme is 64KB, although any multiple of this amount may be available.

After establishing the amount of external memory available, the secondary bootstrap reads the loader into the highest addressed part of the available memory[2]. The secondary bootstrap next allocates a small workspace for the loader to work in and finally transfers control to the loader.

[2]The loader is loaded so that the last byte of its code is a few words away from the end of external memory. This prevents the transputer's instruction pre-fetch mechanism from accidentally causing a reference to non-existant memory.

On entry (see figure 6.5b), the loader has the following information, which has been passed to it by the secondary bootstrap:

- The address of the lowest addressed memory byte which is available for use. This is the INMOS **MemStart** location, which is 80000048_{16} for the T414 and 80000070_{16} for the T800. Memory between this address and the start of the memory map at 80000000_{16} must not be used for any purpose, as these locations are reserved for use by the processor itself.

- The address of the lowest addressed memory byte which is in use by the loader. Memory at addresses lower than this but higher than **MemStart** are available for use.

- Pointers to the channel words of the link connecting this processor to the processor from which it has been bootstrapped.

Throughout its operation, the loader maintains the two memory pointers received from the secondary bootstrap. Memory is allocated from the area between the two pointers, with one or other pointer being moved after each allocation. Thus, low-addressed memory may be allocated by moving the pointer which was initialised to **MemStart**, while high-addressed memory may be allocated by moving the pointer which originally pointed to the start of the loader itself. These two allocation methods can be seen as the two vectors in figure 6.5b; they correspond to the presence or absence of the configuration language **OPT=***area* attribute within a TASK statement.

For example, take the following configuration language statement:

```
task user_task ... opt=code
```

For this task, the low-address pointer would be used to allocate memory for the task's code area, while all other memory areas required by the task would be allocated from the high-addressed end of the free memory area.

The configuration language includes a facility for allocation of the remaining free memory on a processor to a particular task, as follows:

```
task user_task ... data=?
```

This is implemented through the special case mentioned on page 49, whereby a memory area size of −1 is not immediately allocated, but is deferred until the tasks are finally started on receipt of a GO command. At this point, the unused memory on the processor is simply the area between the two allocation pointers.

As well as allocating memory for user tasks, the loader also creates small dynamic data structures for its own purposes. These data structures, which include the vector of 'extra' channels allocated by the loader CHANS command, are always allocated from the high-addressed end of free memory, to avoid their occupying valuable on-chip memory space.

7 decode utility

A separate decoder utility is supplied with Parallel C which takes as its input the binary output file of the compiler, and produces a listing including both the source code and the disassembled machine code for each source line.

An example of **decode**'s output may be found in figure 7.1.

7.1 Usage

The decoder is started by a command of this format:

decode *filename*

Here, *filename* is the name of a binary output file from the compiler. If no extension is typed, **.bin** is assumed.

The decoder then attempts to find the source file, using the source file name given at compilation time, which is stored in the binary file. It applies this name in the context of the current directory when the decoder is run. The decoder should therefore be invoked with the current directory set to be the directory which was current when the file being decoded was compiled.

If **decode** cannot find the source file, it outputs a warning message and produces a disassembly listing without source lines.

The decoder's output is normally sent to the display.

7.2 Features of the decode program

Machine-code instructions are decoded into mnemonics. A complete list of all mnemonics produced can be found in appendix C. The decoder automatically merges **pfix**'s and **nfix**'s with the following opcode. There is full support for all T414 and T800 instructions, including the T800's '**fpu**' operations. Unrecognised indirect instructions are decoded as '**opr** *n*', and unrecognised **fpentry** instructions as '**ldc** *n*; **fpentry**'.

The destinations of **j** and **cj** instructions are shown as addresses in hexadecimal, rather than relative displacements. Calls to external symbols are shown symbolically if possible. The operand fields of all other direct instructions are shown in decimal.

The initialisation values of static data are shown in hexadecimal and ASCII.

The source code contents of **#include** files are not shown, but binary code generated from them is decoded and appears at the right point in the main source file.

7.3 Other languages

The decoder can handle object files which are produced by the INMOS Parallel C, Parallel Fortran and Pascal compilers. If source files are available, the C, Fortran or Pascal source program will be correctly included in the listing.

The INMOS stand-alone occam 2 compiler also generates binary files in this format, and should therefore be decoded correctly, although this cannot be guaranteed. The source programs are not shown, as the occam compiler does not generate the necessary line-number information.

The decoder cannot handle executable (**.b4x**, **.b8x** and **.bt**) files.

```
Transputer DECODE (V1.2) of decodex.bin
ID T4 "occam 2 V2.1" "CC_transputer V2.0"
SC 0
TOTALCODE 140 0
STATIC 2
                          20  0008B              | |
                    00000000  00040              |____|
  5845444F 43454409 00000001  00078              |.___.DECODEX|
                       432E  00084              |.C|
                    00000000  00044              |____|
REF #0,  "_iob"
PATCH LONG 00000044 0 00000001 DATAFIX
  00000001 00000001 FFFFFFFE  00048              |....._____.___|
                    00000000  00054              |____|
     1  #include <stdio.h>
     2  main()
                 6E 69616D04  00086                   |.main|
CODESYMB "main" 00000030
                       BE 60  00030  ajw     -2
     3  {
     4     int a, b;
     5     a = 100;
                       44 26  00032  ldc     100
                          D0  00034  stl     0
     6     b = a/25;
                          70  00035  ldl     0
                       49 21  00036  ldc     25
                       FC 22  00038  div
                          D1  0003A  stl     1
                          B2  0003B  ajw     2
                       F0 22  0003C  ret
                    FFFFFFFF  00058                |....|
  00000000 00000000 01CB01CB  00000              |....__________|
  00000002 00000000 00000000  0000C              |___________.___|
  0000003C 00000044 00000028  00018              |(___D___<___|
  FFFFFFD4 0000000E 00000054  00024              |T___.___....|
PATCH LONG 00000004 MODNUM
PATCH LONG 00000008 STATICFIX
PATCH LONG 0000000C INIT
PATCH LONG 00000010 LIMIT
                    FFFFFFFF  0005C                |....|
  0000003E 00000008 00000030  00060                |0___.___>___|
                    FFFFFFFF  0006C                |....|
     7  }
     8
```

Figure 7.1 Example of output from **decode**

8 `fconfig` flood-fill configurer

There are two types of user task in a flood-fill configured application. One task, referred to as the *master,* divides up the computation to be performed into small *work packets.* The other task, which is known as the *worker,* is replicated all over the network; it accepts work packets originating from the master, performs some computation and sends a reply packet or packets back.

Task images for the master and worker tasks must be created with the `iboot` tool. A configuration file must be created to define the master and worker tasks, and the types of processor allowed in the network.

The flood-fill configurer takes as input the task image files and the configuration file and creates as output a program which will run on an arbitrary transputer network (subject to transputer type constraints defined in the configuration file).

8.1 Running the flood-fill configurer

The configurer has a simple command line:

`fconfig` *configuration-file output-file*

All options are defined in the configuration file. These are discussed in the chapter describing the configuration language (chapter 17).

The configuration file also defines the task image files required by the configurer. If these are not found in the current directory the configurer will use the search path.

8.2 User task protocol

This section describes the protocol used by the user tasks in a flood-filled application. Note that a different protocol may well be used by the router tasks, for example to avoid problems with T414A restrictions on minimum length of messages sent across links.

8.2.1 Master task's ports

The master task has two input ports and two output ports. The input and output ports `master[1]` are connected in the usual way to a file server task such as

iserver (via the protocol filter task **filter**).

The input and output ports **master[0]** are connected to the *router* task. The router task is provided by the flood-fill configurer, and has the function of transporting work packets from the master through the network to idle workers to be processed.

8.2.2 Worker task's ports

Each worker task has one input port and one output port.
These ports **worker[0]** are connected to the part of the routing system which exists on each processing node of the network.

8.3 Packet format

Work and response packets have identical format, consisting of a fixed-length portion and an optional variable-length portion. The two portions of the packet are send as separate messages. Each packet starts with a message containing a 4-byte integer header, as shown in Figure 8.1.

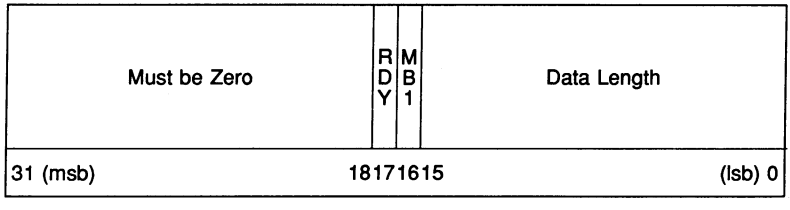


Figure 8.1 Format of Packet Header

The various fields of this 32-bit message are used as follows:

- The least-significant sixteen bits of the message are used to indicate the length of the data block following the header. If the length is zero, no data block follows; otherwise this many bytes of additional data follow as a separate message of that length.
- Bit number 16 (value 00010000_{16}) is always 1.
- Bit number 17 (value 00020000_{16}) is set to 1 to signify that the sending task is *ready*. A worker task can set RDY = 0 to indicate that further response packets will be issued before the next work packet will be accepted.
- Bits number 18–31 are always 0.

9 iboot bootstrap

The **iboot** tool takes as its input a linked C program and produces an executable file that can be loaded down onto transputer and run. The input file will always be the output of the linker.

The **iboot** tool can be considered as a simple configuration program which converts a (non-executable) object file and into a executable code file that will run on a single transputer.

The **iboot** tool also enables a linked *task* to be coverted so that it can be used as input to the 3L general purpose and flood-filling configurers.

9.1 Running the iboot tool

To run the **iboot** tool type:

iboot *filename* {*option*}

where: *filename* is the name of the input file. No default extension is assumed.

option is a list, in any order, of zero or more of the options listed in Table 9.1.

Spaces between the options and the case of letters in parameters are not significant. Options may be specified in any order following the filename.

9.2 What can be made executable

The **iboot** tool assumes that the program is to be run on a single transputer.

The **iboot** tool will not accept the following types of object file as its input:

- object files created by the librarian.
- object files that still have unresolved external references in them (i.e. programs that have yet to be linked).
- object files which contain more than one entry point.

The transputer targets supported by the **iboot** tool are the T414 and the T800. If the transputer target is of any other type then the **iboot** tool will generate an error. The same bootstrap loader is output for the T414 and T800.

Option	Description
C	Produce task image file for input to the 3L configurers.
S *stacksize*	Specify amount of run-time stack for C programs in words.
I	Display information.
L	Load and terminate the tool.
M	Do not produce a code map file.
E	Flip the error action of the bootstrap loader code.
O *output.file*	Specify output file name. Otherwise the iboot tool uses the same file name as the input file with a .bxx extension produced in accordance with the rules in section D.2.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets.	

Table 9.1 iboot options

By default the internal bootstrap loader used by iboot will clear the transputer's halt-on-error flag, which means that if the error flag is set during the execution of a C program the transputer will not halt. If the E option is used then the then the halt-on-error flag will be set.

9.3 Producing task images

If a C program has been linked using the taskharn.txx harness (which means that it is to be used for input to the 3L configurers) then it will be necessary to apply the iboot tool using its C option on the output file of the linker. When this option is used the iboot tools converts the linked C task object file into a file format that the 3L configurers can then use.

When this option is used it is also necessary to use the O option to specify an output file name. The name that is used for the output file should by the same as the input file except that the extension should be changed from .cxx to .b4.

The following is an example showing how a C program is linked using the harness **taskharn.txx** with the full C run-time library which is then converted, using **iboot**, for input to the 3L configurers:

```
ilink taskharn.txx a1, a2, ...  crlt.lib -o o.cxx
iboot o.cxx -c -o o.b4
```

Where *a1, a2, ...* are the object files for the C program and *o* is the output file name (minus extension). The part of the file name extension denoted by *x* is the transputer target for which the C program has been compiled for (**4** for the T414 and **8** for the T800).

9.4 Bootstrap loader interface

The bootstrap loader output by the **iboot** tool loads a linked program in the following way:

The code for the C program is placed as low in memory as possible taking into account the amount of work space required by the special harness that is always linked in when linking a C program (see appendix B). The C program's code is always placed above the work space required by this harness. N.B. The memory reserved by the bootstrap loader for itself will be overwritten by the C program's code and work space when it is started up by the bootstrap loader.

If the harness requires additional work space then the bootstrap loader will reserve memory for this work space and place it just above the C program's code.

The size of the harness work spaces, along with the size of the code for the C program are used by the bootstrap loader to determine the offset, from the start of memory, from which point onwards can be used as the C program's work space. The bootstrap loader determines the size of C program's work space by looking up the environment variable **IBOARDSIZE** before the C program is started.

IBOARDSIZE specifies the size of memory, in bytes, of the transputer board on which the C program is to be executed. (**#** or **$** in front of the memory size indicates a hexadecimal number.)

The memory reserved for the C program's work space is used by the C program for its **static** and global varaibles and for its heap. Also, depending on whether the **s** option of the **iboot** tool is used this work space is also used for the C program's run-time stack.

If the **S** option of the **iboot** tool is used then the amount of memory specified by the option (in words) will be used for the C program's run-time stack. This amount of memory, for the C program's run-time stack, will be allocated by the bootstrap loader below the harness work space and will start from **MemStart** (address 80000048_{16} on the T414 and address 80000070_{16} on the T800).

If the **S** option of the **iboot** tool is not used then the memory reserved for the C program's work space will be used for the C program's run-time stack as well as being used for the C program's **static** and global variables and for its heap. The C program's run-time stack will fall from the top of the work space, the **static** and global variables will be allocated from the base of the work space and the heap will be rising from the top of the **static** and global variables used by the C program.

Figure 9.1 illustrates the memory map of the loaded code as created by the bootstrap loader described above.

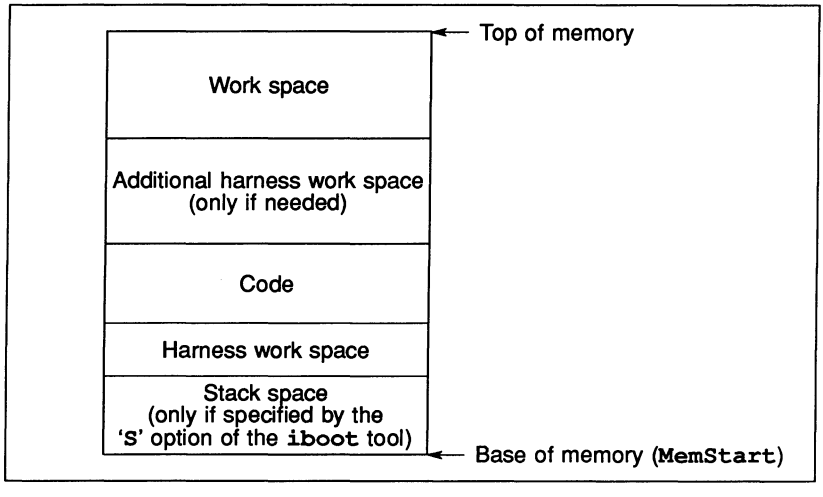


Figure 9.1 Memory map

9.5 Error messages

If an unspecified error message is produced it is possibly due to the file being corrupt.

code not contiguous

The transputer code contained in the input object file is not contiguous, i.e. the code is stored randomly in memory. This can occur if the file has not been linked.

illegal formal parameter specification

List of parameters defined by the main entry point file of the program does not match that required by the bootstrap loader.

illegal link data tag, *tagtype*

An illegal data tag was found in the input object file. This can occur if the file has not been completely linked, or if the file is the output of the toolset librarian.

tagtype can be: STATIC WORD LONG LONGADJ INSTRUCTION COMMON DATASYMB LIBRARY.

illegal option (*chr***)**

An unknown option was specified. *chr* was the invalid option character.

multiple input files specified

More than one input file was specified.

multiple output files specified

More than one output file was specified when using the O option.

multiple stack sizes specified

More than one value for stack size was specified with the S option.

no input file specified

No input file was specified.

no output file specified

No output file was specified after the O option.

no stack size specified

No stack size was specified after the S option, or was set to zero.

unable to close (*value*)

A file on the host system could not be closed. This error can arise if the file does not exist, if the file system is corrupted, if the file is write protected, or if the file system is full. '*value*' is the error result returned by the file system.

unable to open (*value*)

A file on the host system could not be opened. This error can arise if the file does not exist, if the file system is corrupted, if the file is write protected, or if the file system is full. '*value*' is the error result returned by the file system.

unable to read (*value*)

A file on the host system could not be read. This error can arise if the file system is corrupted, or if the file system is full. '*value*' is the error result returned by the file system.

unable to write (*value*)

A file on the host system could not be written. This error can arise if the file system is corrupted, or if the file system is full. '*value*' is the error result returned by the file system.

10 `ilibr` librarian

This chapter describes the library building tool `ilibr`, that collates modules, declarations and files that are related in some way into a single named unit. Files created by the librarian consist of separate modules that can be selectively loaded by programs.

10.1 Introduction

The librarian builds libraries from one or more separately compiled units supplied as input files. The input files may be any object code file produced by the C compiler or files produced by the linker and librarian. In the process it enforces rules about the contents of libraries.

10.2 Running the librarian

The librarian takes a list of compiled (and possibly linked) files and library files, or an indirect file containing a list of such files, and concatenates them to form a single library file. Each file in the input list becomes a selectively loadable module in the library.

To invoke the librarian type:

`ilibr` *filenames* {*option*}

where: *filenames* is a list of input files, separated by one or more spaces.

option is any option from table 10.1.

Option	Description
`I`	Display information.
`F` *filename*	Specify indirect file name.
`O` *filename*	Specify output file name.
`D`	Do not include full debugging.
`L`	Load only. Librarian performs no action.
`X`	Explode library into constituent files.
Options must be preceded by '-' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	

Table 10.1 Librarian options

If an output file is not specified then the name of the first file in the list, or the name of an indirect file, is used instead. Output files are given the `.lib` extension.

10.3 Exploding libraries

The explode option (`X`) allows a library to be disassembled to its constituent files, using the original file names. If the original files are still in the current directory then the original files are overwritten. If an error ocurrs whilst a library is being disassembled constituent files that have already been written are not deleted.

Note: Exploding a library does not delete the library.

The explode option can be used for removing unwanted modules from a library. To do this, explode the library and reinvoke the librarian to restore only the required modules.

For example, suppose you have a library of routines for T414 transputers, called `t4lib.lib`, which contains the files `mod1.bin`, `mod2.bin` and `mod3.bin`. If you decide that you will never use `mod2.bin` you can remove it using the following procedure.

1 Explode the library by typing:

```
ilibr t4lib.lib -x
```

This will re-create the three constituent files `mod1.bin`, `mod2.bin` and `mod3.bin`.

2 Delete the file `mod2.bin` and type

```
ilibr mod1.bin mod3.bin -o t4lib.lib
```

This will recreate the library `t4lib.lib`, but now it will not contain the module `mod2.bin`.

10.4 Removing debug data

When C source is compiled the compiler inserts decoding data for the **decode** tool to use. If you have the source of the object files that make up a library then the **decode** tool can use this data to 'decode' the data of the object files within a library, just as with any other object file created by the C compiler. See chapter 7 for a description of the **decode** tool. The `D` option causes the librarian to remove the decoding data from the object code. This reduces the size of library files.

10.5 Rules for constructing libraries

There are a number of rules governing the construction of libraries.

- A library cannot have a routine (entry point) which has the same name as another routine in the same library if they are compiled for the same transputer type.
- As above but for global variables.

10.6 Library Modules

Libraries are made up of one or more modules. Each file specified to the librarian forms a module. The ordering of modules in a library is unimportant, although modules in a library are implicitly numbered from zero.

A module is the smallest unit of a library that can be separately loaded.

10.6.1 Selective loading

Modules from libraries are selectively loaded by the linker according to the following rules.

1 The transputer target for the module must match the transputer target for the program.

2 At least one routine in the module must be used by the program or by a library that is used by the program.

10.7 Building libraries

This section contains some hints for building libraries.

When building libraries try to keep modules as small as possible. This will ensure that your final program does not contain large amounts of unnecessary code.

In general purpose libraries you can add modules containing the same routines, but compiled for different transputers.

Try to group routines of similar functions into a library. If routines are always used together (for example routines for opening and closing files), group them in a single module.

If your library source references external functions and variables you are recommended to include the referenced code in the library that you are building.

10.8 Indirect files

Indirect files are input files for the librarian that contain a list of files from which the library will be built. To use an indirect file, specify the **F** option and the name of the indirect file. More than one indirect file can be specifided on the command line.

The format of indirect files is as follows.

1 Input file names may be split over any number of lines.

2 Comments may be inserted using the comment symbol '--'. All characters typed on a line after '--' are ignored.

3 Librarian options must appear on a line starting with an option escape character (i.e. '/' or '-') rather than a file name.

Indirect files should have the same name as the library file, but with the `.lbb` extension.

10.9 Error messages

The librarian produces error messages in the standard toolset format. If errors are produced the librarian terminates and no files are produced.

Command line too long (at *string*)

Character limit exceeded on the command line. *string* is the position on the command line where the overspill occurred.

Could not open indirect file *filename*

File system error. The indirect input file *filename* could not be opened.

Expected an option letter, found end of line

Command line error. No option was specified after the option selection character.

Expected filename after -f option

Syntax error. No indirect file was specified after the F option.

Expected filename after -o option

Syntax error. No output file was specified after the O option.

Explode option and *chr* d switch are incompatible

Command line error. The X option and the option given by *chr* cannot be used on the same command line.

Indirect filename *filename* is too long

Syntax error. The maximum length for filenames is 255 characters.

Input file *filename* is not a library

File *filename* is not a library.

Input filename *filename* is too long

Syntax error. The maximum length for filenames is 255 characters.

No input files specified, at least one needed

Command line error. At least one input file must be specified on the command line.

num **Output files specified, only one allowed**

Command line error. Only one output file is permitted and *num* were supplied.

Ran out of memory (at *number* bytes)

Insufficient memory available for the tool to run. This may occur if the libraries involved are very large.

Unable to open input file *filename*

File system error. The input file *filename* could not be opened.

Unable to open output file *filename*

File system error. The output file *filename* could not be opened.

Unknown option letter '*chr***'**

Command line error. An invalid option was specified. *chr* was the first letter of the invalid option.

11 `ilink` linker

This chapter describes the linker tool **`ilink`** which is used to build groups of separate compilation units into object files. The chapter begins with an introduction to the tool, continues with a description of the command and its options, and finally lists linker error messages.

11.1 Introduction

The linker is a tool that builds object code from a list of input files. External references are resolved, and the separate units are combined to produce an object file that can be loaded onto a transputer or tansputer network.

The linker can be driven directly via a command line, or by redirected input through a linker indirect file or standard input. Details of redirected input can be found in section 11.4.

An option to the command allows the linker to run without resolving external references. This allows you to pre-link sub-components of a program during program development.

Input files can be separately compiled program units, or library files. Output from the linker can be used as input to the bootstrap tool **`iboot`**, the librarian **`ilibr`**, and to the linker itself.

11.2 Notes on using the linker

11.2.1 Output files

The linker does not check for coincidence of input and output file names. You should ensure that the same file name is not used for an input file and output file, otherwise the contents of the input file may be lost, replaced with the linked output file.

If an error occurs during execution of the linker any output files are deleted.

11.2.2 Processor type checks

Before linking, the linker checks the processor type of each input code file. If processor types are incompatible, the linker will fail and an error message is generated.

NOTE. The processor type for the linked output file is determined by the first input file on the command line that is not a library file.

11.2.3 Selective loading of library files

The processor type check is not performed on library input files. If the linker finds a module within a library file has been compiled for a non-compatible processor type, the library module will be ignored by the linker. This allows selective loading of library modules based on processor type.

Libraries are also selected for linking on the basis of previous usage. If library modules have not been referenced within the same linking process by other program units, then they are not used.

11.3 Running the linker

To run the linker use the following command line:

ilink {*inputfiles*} {*option*}

where: *inputfiles* is a list of code files generated by the C compiler, or by the linker. If the first file in the list was generated by the linker then it may be necessary to explicitly specify the name of the output file.

option is any of the linker options, given in table 11.1.

To rename the entry point name for an input object file, prefix the following to the input object file:

new.name =

This instructs the linker to change the first entry point name in the associated input object file to *new.name*.

11.4 Redirected command input

The linker can read its command parameter list from three different types of source

- the linker command line
- an indirect file

Option	Description
I	Displays link information.
L	Load linker and terminate.
M	Disables the file map. The default is to produce a map of the linker input files in a file named from the first input file, suffixed with .mxx.
E	Extends linker capacity (two pass operation).
U	Allow unresolved external references.
S	Disables the symbol table. The default is to write the symbol table to a file named from the first input file, suffixed with .sxx.
B (*size*, ...)	Redefine buffer sizes (*size* = decimal numbers).
Q (*symbol*, ...)	Optimize library functions by placing at the beginning of code.
O *output.file*	Specify output file.
F *indirect.file*	Take command input from an indirect file.
Options must be preceded by '-' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	

Table 11.1 Linker options

- standard input, for example, the keyboard.

If the input comes from either an indirect file or the standard input then the input is known as *redirected* input.

11.4.1 Linker indirect files

An indirect file is a file that can contain a list of input object files to the linker, plus linker options. The format for indirect files reflects the syntax of the linker command line, except that the input object file names and linker options can be split over a number of lines.

An indirect file can also contain comments. The start of a comment is denoted by a double dash ('--') and the comment ends at the end of the line.

Indirect files are specified using the F option followed by a file name, as in the following example:

F *indirect.file*

If no file name is specified then the input will be taken from standard input, usually the terminal. The format for input taken from standard input is the same as for an indirect file.

Redirected linker command input may *NOT* itself be redirected. Therefore an indirect file may not refer to another indirect file or standard input. The same applies to standard input, that is, it cannot be redirected to itself or to an indirect file.

11.5 Linker options

Command line options to **ilink** are described in the following subsections.

11.5.1 Option M – disable file Map

The disable file map option disables the production of a file mapping the code that is being linked. This map gives information on the order in which the files were linked (including library files) and the position of each file's code within the address range of the entire code file. The position is given as an addresses in bytes. The map displays information about two categories of input file; separate compilation units, and library modules.

11.5.2 Option E – extend link capacity

Normally the linker makes a single pass over the input files, processing all code and link data in memory, thus reducing the size of code which may be linked. The **E** option forces two passes over the code, and allows larger pieces of code to be linked. If you specify the **E** option, linking will take longer.

11.5.3 Option S – disable Symbol table

This option disables the production of a symbol map of the code. The default if you do not specify this option, is to write a symbol map file with the same name as the output file, but suffixed with the extension **.sxx**.

The symbol map lists all the global code and data symbols that are defined within the program, with their relative offsets from the start of the corresponding code and static data areas.

11.5.4 Option B – change Buffer size

The change buffer size option resets the sizes of the internal buffers used by the linker. To reset buffer sizes, specify new values within the parentheses.

There are seven internal buffers used by the linker, five of which can be changed with the B option, namely, **LINK, DESC, SYMBOL, INIT, REF**. Two other buffers, **FILE** and **CODE**, are fixed and cannot be modified.

New values for the five modifiable buffers must be specified within the parentheses in the correct order, as follows:

B (*LINK, DESC, SYMBOL, INIT, REF*)

The modifiable buffers are described below.

Buffer	Description
LINK	This buffer holds link information from all the object files and is used by the linker to perform any linking operations specified in these object files.
DESC	This buffer holds the description information present in the object file that defines the main entry point of the program being linked. The information contains the formal parameter specification of the main entry point, the entry point offset, and the work space requirements for the main entry point. Also stored in this buffer are the program's processor type and total code size.
SYMBOL	This buffer is used by the linker as its symbol table and is used to store all the code and data global symbols defined in the program being linked.
INIT	This buffer is used by the linker when linking a program that contains one or more C object files. It stores the information that is required by the C components to perform the initialisation of static variables and data segments at run-time.
REF	This buffer is used to store all the external references that are made in each object file being linked (the buffer is re-used for each object file).

Of the non-modifiable buffers, the **FILE** buffer contains the list of the modules being linked (plus the ones specified in the command input, but not linked because of selective loading), and the **CODE** buffer is allocated last, using the space that is not already occupied by the other buffers described above.

Buffer sizes can be displayed by using the information (I) option.

To leave a buffer size unchanged from the default proportional size, leave the position blank. For example, the following command changes the sizes of **LINK** and **SYMBOL** buffers only, leaving the others unchanged from their default proportional sizes:

```
   B (150000,,1000)
or B (150000,,1000,)
or B (150000,,1000,,)
```

If the position is left blank the buffer is set to the default proportional size.

The proportions of the free memory available to the linker are allocated to the buffers as follows:

Buffer	Size	Type	Element
LINK	20%	data used for linking (bytes)	1 byte
DESC	1%	main entry point information (bytes)	1 byte
SYMBOL	15%	global code and data symbols	64 bytes
INIT	1%	modules to be initialised	8 bytes
REF	1%	external symbol references	4 bytes
CODE	—		1 byte

Note that for each element in the **SYMBOL** buffer it has been assumed that the length of each symbol name will be on *average* no longer than 32 characters. This is the average length only, and individual symbol names can be any length up to 255 characters.

CAUTIONS: If the modified sizes for these buffers exceed the total memory space available to the linker it will then not be possible to allocate the **CODE** buffer and an error will be generated. An error is also generated if a buffer size is set to zero or to a negative value.

Note that the free memory available to the linker is treated as a byte vector and the proportions for the buffers are allocated from this byte vector before being converted for use by the buffers.

Calculating memory requirements for a linked program

To determine the amount of memory available for the linker code in bytes, sum the buffer requirements and subtract the result from the total memory size. That is:

LINK.SIZE = *size* **TIMES** 1

DESC.SIZE = *size* **TIMES** 1

SYMBOL.SIZE = *size* **TIMES** 64

INIT.SIZE = *size* **TIMES** 8

REF.SIZE = *size* **TIMES** 4

(*size* will be either the default settings for each buffer, or the values set using the **B** option.)

Using this scheme, the size of the **CODE** buffer will be given by the following:

```
CODE.SIZE  =  TOTAL.SIZE MINUS (
              LINK.SIZE PLUS DESC.SIZE PLUS
              SYMBOL.SIZE PLUS INIT.SIZE PLUS REF.SIZE
              )
```

Some sample buffer sizes for different values of **TOTAL.SIZE** which generate usable **CODE** buffer sizes are as follows :

TOTAL.SIZE	LINK	DESC	SYMBOL	INIT	REF
1 Mbyte	200 000	10 000	2 000	500	500
2 Mbyte	400 000	10 000	4 000	1 000	1 000
4 Mbyte	800 000	10 000	8 000	2 000	2 000
8 Mbyte	1 600 000	10 000	16 000	4 000	4 000

11.5.5 Option Q – optimise symbols

This option allows you to specify which library functions are to be located on the front of the linked code. Functions on the front of the linked code are those which are most likely to be placed in the on-chip RAM.

Placing commonly-used functions on the on-chip RAM will optimise their running

speed, and increase the overall speed of the program.

The library functions to be optimised are specified using the Q option by giving the entry point names of the library functions. The entry name specification can include any character except a space or a comma (,), and must be specifed on a single line. To specify a list of entry names, separate each entry with a comma. If a specified entry name is not used by the program then no optimisation is performed for that entry name.

If no library entry names are specified, then entry names **REAL32OP** and **REAL32OPERR** are optimised, if they are used by the program. These entry names relate to functions that carry out 32 bit real addition, subtraction, multiplication and division.

11.5.6 Order of linking of object files

The object files can be linked in any order, but the processor type for the linked output file is determined from the first input file in the list. For this reason, you should specify the main body of a unit (e.g. **mainent.cxx**) being linked first, so that the transputer type used in the final object is that of the main program unit.

11.6 Error messages

If an unspecified error message is produced it is possibly due to the file being corrupt.

attempted to re-redirect input

> Command input has already been redirected using the **F** option. The **F** option has been given more than once on the command line, for example, by specifying the **F** option from within an indirect file.

code patch over legal code, INSTRUCTION (*value*)

> A code patch specified by an **INSTRUCTION** record has overwritten some valid code, that is, code which not made up of NOP transputer instructions (**pfix 0**). This error generally only happens when there are too few NOP instructions remaining to make a code patch over, and is usually generated when using sequential program inserts. *value* is the code patch offset specified by a **INSTRUCTION** record that generated the error. This can be generated by specifying too small a value of *n* for the '**PC**' option of the C compiler.

expected end of buffer list

The closing parenthesis was omitted when using the B option.

expected end of symbol list

The closing parenthesis was omitted when using the Q option.

expected start of buffer list

The opening parenthesis was omitted when using the B option.

expected start of symbol list

The opening parenthesis was omitted when using the Q option.

file name expected

This error can occur if the filename is omitted when using the O option, or when the prefix '*new.name* =' is used.

illegal buffer size, *buffertype* **(***value***)**

A buffer size less than or equal to zero was specified with the B option. *buffertype* can be LINK, DESCRIPTOR, SYMBOL, INIT, or REF.

illegal character in buffer list (*chr***)**

A non-numeric character was specified in the list of buffer sizes. *chr* is the invalid character.

illegal option (*chr***)**

An invalid option was specified on the command line. *chr* is the invalid option character.

internal buffer overflow, *buffertype*

This message is generated if one of the linker buffer overflows. *buffertype* can be STRING, FILE, LINK, CODE, DESCRIPTOR, REF, INIT or SYMBOL. All but the STRING and FILE buffers can be changed using the B option. The STRING buffer has a maximum capacity of 255 characters; the size of the FILE buffer can be displayed using the linker I option.

multiple entry points, *symboltype* unchanged

The object file referenced by the *new.name=* prefix contains more than one entry point, and records associated with all entry points except the first remain unchanged. *symboltype* can be DESC, ENTRY, NEWENTRY, CODESYMB, ENTRYSYMB, or NEWENTRYSYMB.

multiple MAININIT addresses, MAININIT

More than one **MAININIT** operand to a record was specified. This error can occur when the C run-time library is multiply defined.

no MAININIT address, INIT

No **MAININIT** operand to a record was specified. This error can occur when the C run-time library is undefined.

output file redefined

The o option was specified more than once.

processor type incompatible (*value*)

A module has been compiled for a processor type incompatible with the main the program. *value* is the incompatible processor type.

program entry point undefined

No main program body was specified. The message is also generated if no object file is specified.

reference to undefined symbol, REF (*symbol*)

The external symbol *symbol*, specified by a **REF** record, has not been defined in the program.

selective symbol multiply defined, *symboltype* (*symbol*)

The symbol *symbol* has been defined more than once in the program. *symboltype* can be CODESYMB, ENTRYSYMB, or NEWENTRYSYMB.

symbol multiply defined, *symboltype* **(***symbol***)**

The symbol *symbol* has been defined more than once in the program. *symboltype* can be COMMON, DATASYMB, CODESYMB, ENTRYSYMB, or NEWENTRYSYMB.

unable to allocate buffer, CODE (*value***)**

The CODE buffer could not be allocated. *value* is the invalid size of the buffer, or the amount of space remaining from which to allocate the buffer.

unable to close (*value***)**

File system error. A file on the host could not be closed. *value* is the error tag returned by the host file system.

unable to open (*value***)**

File system error. A file on the host could not be opened. *value* is the error tag returned by the host file system.

unable to read (*value***)**

File system error. A file on the host could not be read. *value* is the error tag returned by the host file system.

unable to write (*value***)**

File system error. A file on the host could not be written. *value* is the error tag returned by the host file system.

12 `iserver` host file server

The host file server, **`iserver`**, provides two functions:

- Control of transputer networks, such as loading programs and resetting processors
- Access to host services for programs running on transputer networks.

12.1 Running the server

To run the host file server use the following command line:

`iserver` {*option*}

where: *option* is any file server option, given in table 12.1

Option	Description
`SB` *filename*	Boot program contained in named file.
`SC` *filename*	Copy named file to link.
`SI`	Produce information messages.
`SL` *name*	Specifiy link address or device name.
`SR`	Reset the root transputer.
`SS`	Serve link (i.e. provide host support to program communicating on link)
Options must be preceded by '-' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Note: `-SB` *filename* is equivalent to `-SR-SS-SI-SC` *filename*	

Table 12.1 File server options

All options are two letters long and start with the letter 'S'. None of these options may be used for program parameters. Any other text on the command line is supplied to programs.

If **`iserver`** alone is typed then the server provides brief help information.

12.1.1 Loading programs

Before a program may be loaded onto a transputer network it must be compiled, linked and made bootable using either the bootstrap tool **iboot** (for single transputer programs), or the configurers **config**, **fconfig** (for multitransputer programs). The file will have a **.bt** or a **.bx*x*** file extension.

The name of the file containing the program to be loaded is specified using the **SB** option. If the file cannot be found an error is reported. When this option is used the board is reset prior to loading the program. When the program has been loaded the server then provides host services to the program.

Note: Using the **SB** option is equivalent to using the **SR**, **SS**, **SI** and **SC** options together.

To load a program onto a board without resetting the root transputer, use the **SC** option. This should only be done if the transputer being loaded has already been reset or has a resident program that can interpret the file.

To terminate the server immediately after loading the program use the **SR** and **SC** options together. The server will then reset the transputer, load the program onto the board, and terminate.

To reset a transputer use the **SR** option.

12.1.2 Specifying link address – option SL

The server contains a default address or device name, depending on the operating system, used when communicating with boot from link boards. This address or name may be changed by the **SL** option followed by the new value. The link addresses must be given in hexadecimal.

13 tc C compiler

This chapter describes the C compiler, **tc**, its facilities and options. It explains how to invoke the compiler, describes the command line options and lists compiler error messages.

13.1 Running the compiler

The compiler takes as input the name of a C text file and compiles the contents into a binary object code file.

Options control the mode of compilation and various compiler facilities, such as disabling the generation of **decode** data.

To invoke the compiler use the following command line:

tc *filename* {*option*}

where: *filename* is the name of the C text file. If you do not specify a file extension, the extension **.c** is assumed. If the filename is omitted the compiler displays brief help information.

option is a list, in any order, of any of the compiler options given in table 13.1.

t4c is equivalent to **tc** with the **T4** option.
t8c is equivalent to **tc** with the **T8** option.

If the compilation is unsuccessful an error message is produced giving information about the file and the line where the error occurred. Compilation error messages are listed in section 13.3.

13.2 Compiler switches

This section describes the switches available to control the behaviour of the compiler. Switches are introduced by the switch character (which is host dependent) and may be typed in any order, before or after the source file specification. Except as noted below, switches and their argument strings are not case-sensitive; that is, lower-case letters have the same significance as the corresponding upper-case letters. This means, for example, that the following two switches would be treated the same:

```
-FBhello.bin
-fbHELLO.BIN
```

Option	Description
C	Check: do not generate object file.
D*macro*	Define *macro* with the value 1.
D*macro=string*	Define *macro* with the value *string*.
FB*filename*	Put binary object output in *filename*.
FL*filename*	Put listing in *filename*.
FO*filename*	Identical to FB.
I	Print the compiler's identification.
I*directory*	Add *directory* to the #include list.
L	Equivalent to FL (obsolescent). A *filename* may not be specified.
M	Include macro expansions in the listing.
PC*n*	Set the number of bytes required for an extern function call.
S	Use single-precision floating-point arithmetic when possible.
T4	Generate object code for the T414 processor.
T8	Generate object code for the T800 processor.
T8A	Generate special object code for the Rev A T800 processor.
U*macro*	Undefine a predefined macro.
V	Verbose: display progress messages.
X	Discard the standard #include list.
Switches and their arguments are not case sensitive, except as noted in section 13.2.	

Table 13.1 C compiler options

The format of the various switches is described using the following notations:

filename The filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 13.2.1 below.

directory The filename, which will be assumed to refer to a directory.

macro Any sequence of characters which is acceptable to the compiler as a macro name.

string Any sequence of characters which is acceptable to the compiler as the value of a macro.

n A decimal integer.

13.2.1 Controlling output files

The **F** switch is used for specifying which output files are to be generated, and their names. Each of the varieties of **F** may be followed by a *filename*, but the complete path name may not be necessary. The compiler supplies defaults, as follows:

- If no extension is given, the compiler supplies a default extension depending on the type of output file: **.lis** for listing files, etc.
- If no filename is given, the filename of the source file is used.
- If no directory specification is given, the directory specification of the source file is used; if the source file specification did not include a directory specification, then the current directory is used.

Switches FB and FO

These switches have the same effect. They instruct the compiler to create an object file in binary format. The default extension is **.bin**.

Notice that if no **FB** or **FO** switches are specified, the behaviour of the compiler is the same as if a **FB** switch were used, with no argument. In order to stop the compiler generating an object file of any kind, the **C** switch must be used (see section 13.2.2).

Switch FL

This switch makes the compiler produce a line-numbered source listing file. The listing file contains any error messages produced by the compiler, as well as the numbered source lines. The default extension is **.lis**.

The listing file produced for the **HELLO.C** program would look like this:

```
Source file: HELLO.C
Object file: HELLO.BIN
Qualifiers:  -T8 -FL
Compiled by: transputer C compiler, CC_transputer V2.0

     1  main ()
     2  {
     3     printf ("Hello, world\n");
     4  }
     5
```

13.2.2 Controlling object code

Switches T4, T8 and T8A

These switches can be used to specify which type of transputer the program is to be compiled for. **T4** and **T8** are only permitted with the **tc** command, as the **t4c** and **t8c** commands supply the appropriate switches automatically, and these will, in fact, appear in the 'Qualifiers:' line of the listing (see section 13.2.1).

The **T8A** switch is valid with the **t8c** and **tc** commands. It makes the compiler generate code to work round a floating-point firmware bug in Rev A of the T800 processor which affects integer-to-real conversions.

Switch S

The C Programming Language[1] states that 'all floating arithmetic in C is carried out in double-precision; whenever a **float** appears in an expression it is lengthened to **double**...'. By default, the compiler follows this rule and evaluates an expression like **a+b**, where **a** and **b** are **float**, by first converting **a** and **b** to **double** and then performing the addition using double-precision floating-point arithmetic.

The **S** switch changes the compiler's behaviour when both operands of an arithmetic operator are **float**. If **S** is used, the operands are not converted to **double**, and the operation is performed using single-precision floating-point arithmetic. This should result in faster program execution, but note that because floating-point arithmetic works with approximations the numerical result of the operation may be different from that obtained normally. Using **S** is not recommended.

Note also that even if **S** is used, floating-point constants are still **double**, and so an expression like **2.0*a** will still be evaluated in double precision (with **a** being converted to **double**). You can avoid this happening by assigning the value **2.0** to a **float** temporary variable beforehand (**two** say) and then writing the expression as **two*a**.

Switch PC*n*

This switch makes the compiler allocate *n* bytes for a call to an **extern** function which must be patched by the linker. The value of *n* determines the maximum displacement of the called function from the point of call. The maximum positive displacement is 2^n bytes. *n* should be in the range 2 to 8. If the **PC** switch is not used, the compiler assumes a value of 6 for *n*, giving a maximum displacement of 16MB. (Similar negative displacements are also allowed, except if $n = 1$ when backward calls do not work). Smaller values of *n* reduce the code size for

external calls (resulting in faster execution) but restrict the total size of the final program image. For example, $n = 5$ allows displacements up to 1MB; $n = 4$ allows up to 64KB. Normally the default value of *n* should be adequate.

Switch C

If this option switch is used, the compiler checks the source file for errors, but does not generate an object file.

13.2.3 Controlling `#include` processing

This section should be read in conjunction with section 14.4, where include file processing is discussed more fully.

Switch I*directory*

This switch adds *directory* to the include list, that is, the list of 'standard places' where the compiler looks for files specified in **`#include`** lines. The *directory* string is assumed to be a directory.

Switch X

This switch excludes the 'standard places' from the include list. Directories added to the include list by means of the **I***directory* switch are not affected, and will still be searched by the compiler.

13.2.4 Macro definitions

This section should be read in conjunction with section 14.3, where predefined macros are discussed.

Switch D*macro* and D*macro=string*

The first form of the **D** switch can be used to define a macro with the value '1'. The second form enables the user to define a macro with the value '*str*'. These definitions are done before the compilation of the program. For example:

```
T8C -dDEBUG -Dhelp=3 -dJOE=Jim CATS
```

This is equivalent to coding the following lines at the top of the program **cats.c**:

```
#define DEBUG 1
#define help  3
#define JOE   Jim
```

Notice that the macro names and their values are case sensitive. If there are any syntax errors in the definitions, these are reported on the display and included on the listing (if any) in the usual way.

Switch **U***mac*

This switch undefines a predefined macro — see section 14.3 for a discussion of these. This means, for example, that the following switch:

```
T8C -U_transputer CATS
```

is equivalent to coding the following line at the top of **cats.c**:

```
#undef _transputer
```

Once again, the name of the macro is case sensitive.

13.2.5 Information from the compiler

Switch **I**

This switch makes the compiler display a line containing its identity and version. Please quote this information in any correspondence about the compiler.

Switch **M**

This switch causes the expanded form of lines containing macros to be written to the listing file. By default, macro expansions are not listed. If a **M** is used without a **FL**, the latter is assumed. An example of a listing file containing macro expansions is shown below.

```
Source file: MACRO.C
Object file: MACRO.BIN
Qualifiers:  -T8 -FL -M
Compiled by: transputer C compiler, CC_transputer V2.0
     1  #define SEVENTEEN PLUS(TEN,SEVEN)
     2  #define PLUS(a,b) ((a)+(b))
     3  #define TEN 10
     4  #define SEVEN 7
     5
     6  main()
     7  {
     8    printf("seventeen = %d\n", SEVENTEEN);
     8"   printf("seventeen = %d\n", PLUS(TEN,SEVEN));
     8"   printf("seventeen = %d\n", ((TEN)+(SEVEN)));
     8"   printf("seventeen = %d\n", ((10)+(SEVEN)));
     8"   printf("seventeen = %d\n", ((10)+(7)));
     9  }
```

Notice that the compiler does not list the definitions of the predefined macros, or of macros defined by **D** switches.

Switch V

Makes the compiler produce additional messages on the standard output stream indicating how far compilation has progressed. By default, only error messages are written to the standard output stream and no messages are produced if no errors are detected.

Typical messages generated by use of the **V** option are:

```
123 statements analysed; no errors detected
Code generation complete: starting object file
generation
Object file complete: deleting scratch files
```

13.2.6 Obsolescent switches

The **L** switch is provided for compatability with earlier INMOS C compilers. It is the equivalent of **FL** except that it does not accept arguments and so cannot redirect the output listing file.

13.3 Compiler error messages

This section shows how error conditions are reported by the compiler, outlines ways of dealing with errors detected by the compiler and lists the error messages which may be produced by the compiler along with examples showing how they might come about.

13.3.1 Compiler error message format

This section describes the error reports displayed by the compiler when it detects errors in the program it is trying to compile. Errors which can be detected by the compiler in this way are the easiest to correct. If an attempt is made to compile a program which does not obey all of the rules of C, the compiler will display a message indicating the nature of the fault and showing where in the program the error was detected. For example, in the following program the brackets which must surround the expression following the keyword **while** have been omitted:

```
main()
{
   int i = 0;

   while i++ < 10
      printf("hello, world\n");
}
```

The compiler will discover the error and display a message like this:

```
*"prog.c", line 5:    while i++ < 10
                                ^
( expected
```

The upward arrow character '^' points to the place where the error was found.

Notice the format of the message: all of the messages which the compiler can produce appear in a similar form. The first character in the message is a marker which indicates how bad the error was — an asterisk '*' is the normal sort of error; it means that the compiler has detected a fault but is able to continue trying to compile the rest of the program. The other markers which can appear are described later.

Following the marker character there is the name of the file in which the error has occurred, followed by the line number in the original C program at which the error was detected; here the error is on line five. Wherever possible the compiler displays the text of the offending line after the line number, as in the example program, but because the original text is stored in a fixed size memory area, this cannot always be done. If the source text is no longer in memory it is omitted from the error message.

The general form of compiler messages is therefore:

marker"*filename*", **line** *line-number*: *source-text*
^ *message-text*

Here '^' points to the part of the *source-text* in error, *message-text* is a brief explanation of the fault, and *marker* may be any of '*', '?' or '!'.

Marker	Meaning
'*'	Error: compilation continues
'?'	Warning: a part of the program is strictly correct, but is dubious in some way. For example, if some part of a program can never be reached.
'!'	Fatal error. Compilation cannot continue after a fatal error. Fatal errors indicate either that a program is too large or complicated to be compiled in the amount of memory available or that there is a fault in the compiler itself which makes it unable to compile this program. Section 13.3.4 gives information about what should be done if any particular fatal error is reported.

The line number information can be used to locate the incorrect line quickly with a text editor even when a program contains **#include** statements, because each **#include** counts as a single line, no matter how many lines the included file contains. Look at two C program files called **main.c** (figure 13.1) and **error.h** (figure 13.2).

```
#include "error.h"

main()
{
   while count++ < 10
      printf("hello\n");
}
```

Figure 13.1 File **main.c**

```
/* error included text */

auto int count;
```

Figure 13.2 File **error.h**

If we compile **main**, we will get the following error messages:

```
*"error.h", line 3: auto int count;
                              ^
an external data definition may not have storage
            class "auto"

*"main.c", line 5:    while count++ < 10
                            ^
( expected
```

These messages indicate that in line three of the included file **error** the declaration of **count** is not allowed (because only **static** or **extern** declarations are allowed at the outermost level of a program), and in line five of **main** an opening bracket must follow the keyword **while**.

13.3.2 Fixing errors detected by the compiler

This section contains information about how the compiler handles errors in the program which it is trying to compile. This information should make it easier to understand the messages displayed by the compiler, and so make it easier to fix incorrect programs.

The compiler can detect two classes of error: errors in the *form* of a program such as missing semicolons, misspelt keywords, etc. and errors where an identifier of a particular type is used in the wrong context, such as attempting to multiply a **struct** variable by a **float** value or use an identifier that has not been declared.

Errors of form (*syntax errors*) are detected when the compiler discovers that the piece of program it is reading does not fit in the context of the part of the program it has read already. When this happens the compiler displays a message and starts reading on from the point of the error, ignoring everything until it finds a symbol which *could* fit in at this point in the program; compilation then continues as though there had been no error.

Because the compiler may ignore vital parts of the program (like declarations) in recovering from an error, the best policy when fixing errors reported by the compiler is to deal with them one by one, starting with the first. Look at the part of the program indicated by the error message and try to find out what is wrong with it, then fix the problem and recompile the program. If errors are dealt with sequentially like this, you will not waste time hunting for spurious errors caused by the compiler skipping over some declarations and then complaining about 'undeclared identifiers' in the rest of the program. Look at the example below,

where a comma in a declaration has been mistyped as a dot.

```
main()
{
   int length . breadth;

   length = breadth ;
}
```

This compiler will display the following messages:

```
*"ex.c", line 3:    int length . breadth;
                                ^
; expected

*"ex.c", line 5:    length = breadth ;
                             ^
"breadth" not declared
```

The first message indicates that a semicolon or comma must follow each identifier in a declaration; a dot is not allowed. Because the compiler has skipped the declaration of **breadth** in order to get back in step with the program, **breadth** appears not to be declared in line five resulting in the second error message.

If you correct this program as suggested above, by starting with the first error, fixing it and then recompiling the program then you will never have to worry about fixing the second error: it will go away automatically when the first error is fixed.

In certain cases the logic of the compiler will result in the same error being reported more than once. The remedial action here is simply to fix the error and ignore the duplicated messages.

13.3.3 Compiler control lines

The current version of Parallel C follows the common convention that compiler control lines must start in the first character position on the line. If the '#' character is not the first character on the line then the compiler will report an error which is unlikely to be related to the compiler control line as in the following example:

```
      1  main()
      2  {
      3     #if 1
      4     #else
      5     #endif
      6  }
*     3     #if 1
            ^
} expected
```

13.3.4 List of error messages

The messages listed here may be issued by the compiler while a program is being compiled.

Some messages contain special sequences like *item-1*, *item-2* etc. These do not appear in the actual message displayed by the compiler, rather they are replaced by appropriate text from the program. For example, take the message:

```
"item-1" not declared
```

If it is the identifier `"foo"` which has not been declared, the message actually displayed will be:

```
"foo" not declared
```

Where feasible, the description of each message includes a sample (meaningless) program which causes the message to be generated during compilation.

Program errors

This section gives a list of messages which may be generated by the compiler as a result of errors in the source program or limitations imposed by the compiler.

a bit-field must have an integer type

C allows an implementation to restrict the type of bit-fields. Parallel C imposes the restriction that all bit-fields must have a type which yields integer values.

```
error()
{
   struct thing { float wee:9; };
}
```

a compiler-control (#) line may not begin with "*item-1*"

Compiler control lines are introduced by a hash character, '#', followed by a keyword. This message indicates that a valid keyword has not been found.

```
error()
{
#?rubbish
}
```

a constant integer expression is required here

This message indicates that an identifier or a string literal has been found in a context which requires a constant integer expression.

```
error()
{
   int a[12];    /* right */
   int b[a];     /* wrong */
}
```

a field may not exceed 32 bits

C limits the size of a bit field to the size of an `int` which on the transputer is 32 bits long.

```
error()
{
   struct thing { int huge:999; };
}
```

a function result of type "*item-1*" is not allowed

This message is generated when an attempt is made to define a function which returns an array or a function.

```
error()
{
   int f()[17]     /* f cannot return an */
                   /*  array of 17 items */
}
```

a parameter declaration may not be initialized

A parameter declaration simply gives information about the sort of value being passed as a parameter, the actual value of the parameter being given when the function is called. This message could be the result of placing the declaration of what should be local variables before the opening '{' of the function.

```
error(x)
int x = 3;
{
}
```

a parameter may not have storage class "*item-1*"

This message indicates that a parameter type specification has attempted to give a parameter an invalid storage class. This can be the result of confusing parameter specifications with local variable declarations.

```
error(x)
static int x;
{
}
```

"*item-1*" already defined

This message is issued when an attempt is made to redefine the tag of a `struct` or `union`.

```
error()
{
   struct thing{int a,b;};
   struct thing{float c,d;};
}
```

an empty enumerator list is not allowed

The list of enumeration constants in the declaration of an enumerated type must not be empty; there must be at least one enumeration constant.

```
error()
{
   enum transparent {};
}
```

an empty structure is not allowed

Every `struct` must have at least one member; it is not possible to have structures with no members.

```
error()
{
   struct empty {};
}
```

an external data definition may not have storage class *"item-1"*

Variables declared outside a function may only have a limited selection of storage classes. This message indicates that such a declaration has a prohibited storage class.

```
register int r;
error()
{
}
```

"item-1" **and** *"item-2"* **are incompatible operand types for the** *"item-3"* **operator**

This message indicates that an attempt has been made to apply the given operator to operands of inappropriate types.

```
error()
{
   int z;
   struct {int a,b;} x,y;
   if (x<y) z=0;
}
```

array dimension table full

There is a global limit on the overall complexity of array and structure declarations. This fatal error message is issued when the program exceeds this complexity. The remedial action is to simplify the program or split it into two or more separate files.

It is not feasible to give a simple example of a program which would generate this fault.

attempt to assign address to short or char

The address of an object is a value which will almost certainly be too large to be assigned to a **short** or **char** sized object. While this is not prohibited it will result in the pointer value being converted into an **int** and then the more significant bits being truncated. As it is very likely that this effect will not be what was intended the compiler will issue this warning.

```
warning()
{
   int v;
   static char text = "hello"  /* wrong */
   short s = &v;               /* wrong */
   char x = &v;                /* wrong */
}
```

attempt to divide by zero

The compiler has detected an attempt to divide by zero. Note that this can happen in two distinct places in the compiler: in a context where the result of the division is needed during the compilation, or when the value is not strictly needed until the compiled program is executed but the compiler is attempting to simplify the expression. This particular error message is a result of a division by zero in the first case.

```
error()
{
#if 1/0
#endif
}
```

The second case gives rise to a '**Zero divide**' message which is described in section 13.3.4.

auto/register arrays and structs may not be initialized

This message is issued when an attempt is made to initialize arrays or structs with storage class **auto** or **register**.

```
error()
{
   int vector[3] = {1,2,3};
   struct {int a,b;} s = {100,200};
}
```

bad escape code "'"*item-1***'**

The given escape code has been detected in a string or character constant but has no meaning. This is commonly caused by including an escape character, '\', in a string without using another escape. In the following example the fragment '\ ' should be written '\\ '.

```
error()
{
   printf("cannot open \ ");
}
```

& before array or function ignored

When used on its own as an operand in an expression, the identifier of an array or function represents the address of that array or function. This message indicates that an '&' operator has been ignored when it has been used redundantly on an array or function.

```
error()
{
   int ad;
   int a[12];
   ad = &a;
   ad = &error;
}
```

both operands for pointer "-" must have the same type

When '–' is used to obtain the difference between two pointer values the types of the two pointers must be identical.

```
error()
{
   int x;
   float *f;
   double *d;
   x = d-f;
}
```

case "*item-1*": already defined

This message indicates that a switch statement contains two or more actions defined for a particular case.

```
error()
{
   int x;
   switch (x) {
case 1 : x = 100;
case 1 : x = 200;
   }
}
```

"case" and "default" are only allowed inside a switch statement

The keywords **case** and **default** are reserved for use within **switch** statements and may not be used in any other contexts. The message frequently indicates that a switch statement has been prematurely terminated or has not been recognised because of a syntactic error.

```
error()
{
   int x;
   case 1  : x = 0;
   default : x = 1;
}
```

'item-1' **character not allowed here**

The given character is either a control character or the character grave (' `'). Such a character may only be used in very restricted circumstances. The most likely causes for this error are typing a grave when a single quote character was wanted or accidentally inserting control characters into the source file.

```
error()
{
   `grave error;
}
```

closing '>' expected

An include statement has attempted to specify a search of standard places only by enclosing the file name in '<' and '>'. The message indicates that the compiler has found the opening '<' but not the closing '>'. One cause of this error is not pressing the shift key when typing the '>' character and getting '.' instead.

```
#include <thing.
error()
{
}
```

constant integer expression required here

This message is generated when the compiler is expecting an expression which can be evaluated at compile time to give an integer value but no such expression can be recognised.

```
main()
{
   int x[1.5];
}
```

constant integer value too large

This message indicates that overflow occurred while processing an integer constant. Currently this is only detected in the case of octal or hexadecimal constants.

```
error()
{
   int x = 0x123456789;  /* > 32 bits */
}
```

corrupt syntax tree

This message indicates that an error has occurred in the compiler itself.

It is not feasible to give a simple example of a program which would generate this fault.

declarator may only contain a single formal parameter list

Following a function identifier definition there may be at most one list of formal parameter identifiers enclosed in parentheses. This message indicates that two or more such lists have been found.

```
error(x)(y)
int x, y;
{
   x = y;
}
```

declaration syntax fault

This message is generated whenever a declaration has been specified incorrectly. As there are many reasons for the error it is best to examine the declaration at the point indicated by the upward arrow in the compiler's report. If the cause of the error is not obvious the formal definition of the syntax of the relevant declaration should be checked.

```
error()
{
   int a,;
}
```

#endif/#else without matching #if

The compiler control lines **#else** and **#endif** must follow a matching **#if** control line.

```
error()
{
#else
#endif
}
```

#endif pending at end of file

This message is issued when the end of the source file has been reached and no **#endif** has been found to match a previous **#if**.

```
error()
{
#ifdef flag
}
```

end of file in argument list of macro *"item-1"* **at line** *"item-2"* **(missing ")"?)**

This message indicates that the end of the source file has been found before the compiler has found the closing parenthesis of a reference to a macro.

```
error()
{
#define thing(x) 1-x
   int a;
   a = thing(x
}
```

expanded macro line too long

This fatal error message indicates that the compiler's macro expansion area has become full and further processing is impossible. The usual cause for this is a recursive macro as in the following example.

```
error()
{
#define rubbish rubbish+1000
   rubbish
}
```

item-1 **expected**

The given item is expected at the indicated point in the program. Note that there may be several different items which would fit but the compiler will only indicate the most likely one.

```
error()
{
   int a    /* semicolon omitted */
}
```

expression expected

The compiler expects to find an expression at the indicated point in the program.

```
error()
{
  case {
  }
}
```

expression of type "*item-1*" cannot be used as a function

This message is issued when an expression which is not a function is applied to an argument list.

```
error()
{
   17(0);     /* 17 isn't a function */
}
```

expression of type "*item-1*" used instead of "int"

This message is given when an expression of a type other than '`int`' has been used in a context which requires a condition.

```
error()
{
   float f, g;
   if (f) g = 0;
}
```

expression syntax fault

An expression has been incorrectly specified. This is usually the result of a typographical error with operators. Check the form of the operator you require and correct the expression accordingly.

```
error()
{
   int a,b,c;
   b = 12;
   c = 5;
   a = b+&c;     /* rubbish */
}
```

format is #include "*file*" or #include <*file*>

This message indicates that the file reference in an **#include** compiler control line has not been specified correctly. The two acceptable forms are **#include** "*file*" which will search for *file* starting in the current directory and then searching the standard place, and **#include** <*file*> which only searches the standard place.

```
error()
{
#include something
}
```

{ function-body } expected here; could be missing ; after) above?

The opening brace, '{', of a function body could not be found following the function heading. N.B. this message is unfortunately very common, as it is easy to make a syntax error which makes a function declaration look like a function header to the compiler.

```
extern f()  /* ; omitted */
int a;
double d;
g()
{
   a  = 17;
}
```

function declarator required before '{'

This message was generated by earlier versions of Parallel C and should no longer be encountered. It was issued when a declaration appeared syntactically to be a function declaration but did not have the type 'function returning ...'.

identifier expected

This message indicates that the context demands an identifier but something else has been found.

```
error(1)
{
}
```

identifier or {enum-list} required after 'enum'

Following the **enum** keyword there must be either an identifier or a list of enumeration constants enclosed in parentheses. Note that the **enum** construct is not in any case supported by this version of Parallel C.

```
error()
{
  enum colour {red, yellow, green, blue}/*right*/
  enum;                                 /*wrong*/
}
```

identifier or {struct-decl-list} required after 'struct'/'union'

The keywords **struct** and **union** must be followed by either an identifier or a declaration of the contents of the structure or union.

```
error()
{
   struct;
   union;
}
```

implementation restriction: pointers to functions cannot be initialized

This message was issued by previous versions of Parallel C. The current versions of the compiler do not have this restriction and so the message should never be generated.

implementation restriction: "sizeof" not allowed in this context

The current implementation of Parallel C does not permit the use of the operator **sizeof** in a constant expression.

```
error()
{
   int x;
   int a[sizeof(x)];
}
```

include stack underflow

This message indicates a malfunction in the compiler itself. The only remedial action is to attempt to simplify the include file structure of the program.

It is not feasible to give a simple example of a program which would generate this fault.

"item-1" **incompatible with type** *"item-2"*

This message indicates that an incompatible combination of type specifiers has been given in a declaration.

```
error()
{
   long char x;
}
```

initializer string longer than array

The string constant which has been used to initialize an array of `char` contains more characters than there are elements in the array. Note that there is always an invisible '`\0`' character on the end of every string constant so that the number of characters it contains is one larger than it may appear to the casual reader. The message is simply a warning that the string constant will be truncated for the purposes of initialization by ignoring one or more of the rightmost characters.

```
warning()
{
   static char x[3] = "1234";
}
```

internal error

This message indicates that an error has occurred in the compiler itself.

It is not feasible to give a simple example of a program which would generate this fault.

"*item-1*" is not in the parameter list of "*item-2*" and so may not appear here

This message is generated when a parameter specification following a function heading attempts to describe an identifier which has not been given in the parameter-list of the function. This can be caused by the incorrect placement of local variable declarations before the opening brace ('{') of the function body compound statement, or by misspelling an identifier in the function heading or in its declaration.

```
error()
int p;
{
   p = 0;
}
```

ISO code *item-1* illegal in strings

A string or character constant contains an illegal character whose ISO (ASCII) code value is *item-1* (decimal). This is most commonly caused by attempting to include control characters, such as newline, in strings explicitly rather than by means of the `\n` escape sequence. Correct the program by replacing such explicit characters with their escape code equivalents.

```
error()
{
   printf("line 1\nline 2");    \* correct *\
   printf("Two bells \07\07"); \* correct *\
   printf("line 1
line 2");                      \* wrong   *\
}
```

label "*item-1*" has already been defined in this function

Within any function a particular label may only be used once as the prefix to a statement. This message is the result of using the given label as a prefix on two or more statements.

```
error()
{
   int x;
here: x = 1;
here: x = 2;
}
```

label "*item-1*" is used in function "*item-2*" above but is not defined there

The named function contains a **goto** statement or assembly-language statement which references a label which has not been attached to any statement within the function. Note that C restricts the use of the **goto** statement to transfer control within a function; it is not possible to use **goto** to transfer control out of a function.

```
error()
{
    goto somewhere;
}
```

Note that unknown identifiers used in **asm** statements are implicitly declared as labels in case they may be forward references to real labels. This means that misspelling identifiers in **asm** statements may result in this message.

left operand of "*item-1*" is not an lvalue

An *lvalue* is an expression referring to a manipulable region of storage. This message indicates that the given operator demands an lvalue but its operand does not refer to appropriate storage.

```
error()
{
    int x;
    &x = 12;
}
```

left operand of '.' must be a structure

The operator '**.**' is used to select a particular field from a structure. This message indicates that '**.**' has been used to select a field from an object which is not a structure and therefore cannot have any fields to be selected.

```
error()
{
    int x;
    x.x = 0;
}
```

macro expansion stack full

During macro expansion, the expanded macro with actual parameters substituted for formals is held in a 4KB buffer. This fatal error message is issued when this buffer has been filled.

macro text store full

This fatal error message is issued under two circumstances:

1 The body of a **#define** macro is too long (currently the limit is 1023 characters).

2 When expanding a function-like macro the size of the actual arguments exceeds 1023 characters.

It is not feasible to give a simple example of a program which would generate this fault.

missing)

A right parenthesis has been omitted from an expression or parameter list. This is commonly caused by mismatching parentheses in complex expressions or by forgetting to depress the shift key when typing ')' and getting a different character, '9' in the following example.

```
error()
{
   int a;
   a = 2*(a+19;
}
```

missing operand

An expression contains an operator which has not been given a required operand.

```
error()
{
   int a;
   a = -);    /* no operand for the "-" */
}
```

"item-1" **must be within a loop**

The keywords **break** and **continue** are used to control the execution of a loop (**for**, **while**, or **do**). This message indicates that **break** or **continue** has been found but not within the body of a loop.

```
error()
{
   break;
}
```

'%' must have integer operands

The modulus operator, '%', returns the remainder from the division of its operands, both of which must yield integer values.

```
error()
{
   int n;
   n = 123 % 4.5;
}
```

not a constant

This message is generated when a constant value was expected but something else was found.

```
error()
{
   int x;
   switch (x) {
case x : x = 0;
   }
}
```

"item-1" **not declared**

This message indicates that an identifier has been used without having been declared previously. Note that in C the case of letters in identifiers is significant. The error can be the result of mis-spelling an identifier or simply forgetting to declare it.

```
error()
{
   int Thing;
   thing = 0;
}
```

number of macro actual parameters does not agree with definition

This message is generated when a reference to a macro has been given a number of parameters which is different from the number of parameters specified when the macro was defined.

```
error()
{
#define mac(x) x+1
   int a, b;
   b = 0;
   a = mac(b);      /* right */
   a = mac(a, b);   /* wrong */
}
```

one or more #endif lines inserted before extra #else here

This message is generated if an **#else** compiler control line is found while the compiler was expecting an **#endif** control line. The compiler assumes that the **#endif** for a previous **#else** has been omitted and that the **#else** it has just found belongs to an enclosing **#if** statement.

```
error()
{
#if 1
#else
#else    /* no corresponding #if */
#endif
}
```

only "extern" or "static" functions are allowed

This message results from attempting to define a function with a storage class other than **extern** or **static**.

```
register error()
{
}
```

only one "default" statement is allowed per switch statement

The **default** statement prefix is used to specify the action to be taken in a **switch** statement when an actual case has not been explicitly handled by a **case** label. It follows that a second or subsequent **default** must be in error.

```
error()
{
   int x;
   switch (x) {
      default : x = 1;
      default : x = 2;
   }
}
```

operand of -> or unary * must have pointer type

The left-hand operand of the operators '->' and unary '*' must be objects which have a pointer type. This message indicates that the given operator has been given an operand which has not been defined to be a pointer.

Unfortunately this message also results from errors in arrays. This is because the C definition of array accesses is in terms of the unary '*' and pointer '+' operators.

```
error()
{
   int a;
   struct point {int x,y;};
   struct point b;
   int x[10];
   int p,q;
   *a = 983;
   b->x = 983;
   a[p][q] = 983;
}
```

operand of unary "*item-1*" must be an lvalue

An *lvalue* is an expression referring to a manipulable region of storage. This message indicates that the context demands an lvalue but the expression given does not refer to appropriate storage. Note that a pointer value (yielded by '&') is not an lvalue.

```
error()
{
   int x;
   x = &12;
}
```

"*item-1*" operator not allowed in a constant-expression

Only a limited number of operators may occur in expressions which must yield constant values at compile time. This message indicates that such a constant expression contains a prohibited operator.

```
error()
{
   int a[(1,2)];
}
```

original and result types for cast must be scalar or pointers

A cast may not involve array or function types, although pointer to array and pointer to function types are permitted.

```
error()
{
    (int []) 0;  /* can't cast to an */
                 /*      array of int */
}
```

"item-1" **previously declared as** *"item-2"* **may not be redeclared as** *"item-3"*

This message results from attempting to declare an object when it has already been declared.

```
error()
{
  int a;
  float a;
}
```

sizeof operand must be a type name or unary expression

The **sizeof** operator takes as its argument something which either has or implies a requirement for a number of bytes of storage. It is this number which is returned as the result. The message indicates that the argument given to **sizeof** is not associated with a quantity of storage.

```
error()
{
   int a;
   a = sizeof(else);
}
```

statement expected here

A statement which controls another statement has been specified without any statement to be controlled.

```
error()
{
   int x;
   if (x) else;
}
```

storage class incompatible with a previous declaration

This message is issued when a declaration contains more than one storage class specification.

```
error()
{
   static extern int x;
}
```

string constant too long

The Parallel C compiler limits the size of string constants to 255 characters. This message is generated if a string constant is found with more than 255 characters. This error is often caused by omitting the closing double quote of a string constant, with the result that a section of program text is interpreted as part of the string. Because this can lead to total confusion later the error is considered fatal and compilation is abandoned.

```
error()
{
   printf("12345678901234567890123456789012\
34567890123456789012345678901234567890\
12345678901234567890123456789012345678\
90123456789012345678901234567890123456\
78901234567890123456789012345678901234\
56789012345678901234567890123456789012\
34567890123456789012345678901234567890");
}
```

struct tag "*item-1*" not declared yet

This message results from an attempt to declare a structure before the structure tag has been declared. It is only possible to declare *pointers* to structs which have not yet been defined.

```
error()
{
  struct x p;
}
```

structure of this type has no *"item-1"* field

The operator '.' has been used to select the named field from a structure but the structure does not contain a field with that name.

```
error()
{
    struct coord {float x, y;};
    struct coord point;
    point.z = 0;
}
```

switch expression must have integer type

The expression used to select a particular case in a **switch** statement must yield an integer value.

```
error()
{
    float x;
    switch (x) {
    case 1 : x = 0;
    }
}
```

syntax error in compiler-control (#) line

This message is generated when part of a compiler control line cannot be understood. The error could be caused by terminating an **#include** control line with a semicolon.

```
error()
{
#include <fred>;
}
```

too many initializers for object of type *"item-1"*

This message indicates that a declaration has included an initialization which contains more items than the object being initialized.

```
static float n = {1,2}; /* n can only take*/
                        /*  1 value not 2 */
error()
{
}
```

too many macro parameters

The compiler currently limits the number of parameters in any macro to 32. This fatal error message indicates that the limit has been exceeded.

```
error()
{
#define silly(P1,P2,P3,P4,P5,P6,P7,P8,P9,\
          P10,P11,P12,P13,P14,P15,P16,P17,\
          P18,P19,P20,P21,P22,P23,P24,P25,\
          P26,P27,P28,P29,P30,P31,P32,P33) 0
}
```

too many names

The program has used so many identifiers that the compiler's dictionary has become full leaving no space for new identifiers. It may be possible to solve the problem by replacing some long identifiers with shorter ones or by splitting the file being compiled into two or more files which can be compiled separately.

If neither of these alternatives works it will be necessary to compile the file on a system with more memory.

It is not feasible to give a program which demonstrates this error!

too many nested #include files

This message results from an attempt to include a file which needs a file to be included which needs a file to be included and so on to the limit of the compiler's ability to open files (currently three include files open at once). One possible cause of this would be a file attempting to include itself! Remedial action is to reduce the depth of include file nesting, perhaps by textual substitution of one of the more deeply-nested files.

It is not feasible to give a simple example of a program which would generate this fault.

type "*item-1*" may not be "unsigned"

The keyword **unsigned** may only be applied to a restricted selection of type verbs. In particular, **float** and **double** may not be specified as unsigned.

```
error()
{
   unsigned int    a;  /* right */
   unsigned short  b;  /* right */
   unsigned char   c   /* right */
   unsigned float  d;  /* wrong */
   unsigned double e;  /* wrong */
}
```

type "*item-1*" not allowed

This message is the result of attempting to declare an array of objects which cannot be combined into arrays, functions for example.

```
error()
{
   static int *x[12]();
}
```

unary "*item-1*" may not have an operand of type "*item-2*"

This message indicates that the given unary operator has been applied to an operand of the given type when such an operation is not permitted.

```
error()
{
   float f;
   f = ~f;    /* logical negation only */
              /* applies to integers   */
}
```

union type objects may not be initialized

This message indicates that an attempt has been made to initialise an object which is a **union**.

```
union x {int p; float q;};
union x thing = 12;
```

unexpected colon in statement context

This message is issued when a colon is found in an unexpected position. One reason for this error is accidentally typing a colon at the end of a statement rather than a semicolon.

```
error()
{
   int x;
   x = 0:
}
```

unexpected end of file (perhaps missing " or */ symbol?)

This message results from the compiler reaching the end of the source file when it was expecting more input. A common cause of this error is omitting or mistyping the closing '*/' of a comment or omitting the closing double quote of a string constant.

```
error()
{
   /* This comment is never terminated
        because wrong slash used here *\
}
```

unimplemented feature *item-1*

The program contains a feature which is correct C but which has not been implemented in the version of the compiler being used. The only remedial action is to recast the offending section of the program in a different form.

```
error()
{
   enum hue {red, yellow, green=20, blue};
   /* enum has not been implemented yet */
   enum hue col;
   col=green;
}
```

unknown size

This messages indicates that a statement requires the size of an object to be known while that statement is being compiled but the actual size cannot be determined.

```
error()
{
   int x;
   x = sizeof(void);
}
```

value out of range

This message indicates that an initializing value is outside the range of values that can be stored in the bit-field being initialized.

```
error()
{
   static struct {int i:3; } x = {255};
}
```

System errors

This section gives a list of the error messages that may be generated during compilation as a result of the interaction between the compiler and the operating system. These messages give information about errors associated with the compilation process itself and are independent of the C language and the general form of source programs.

All of these messages are introduced by the phrase: '**Fatal Error --**' and result in the termination of the compilation.

cannot open #include file "*filename*"

An **#include** compiler control line has referenced a file which cannot be accessed. Check that the filename has been spelled correctly and that it exists in the relevant directory

Note that the filename given in the error message is the full path and name of the final file the compiler attempted to access; forms of **#include** which require the compiler to search two or more directories will not necessarily report the same filename string as specified by the programmer.

/D and/or /U qualifiers are too long

As the compiler scans any **D** or **U** switches typed by the user, it converts them into **#define** and **#undef** compiler control lines, and stores them in an internal buffer. If this buffer is filled up, the compiler reports this error. In practice, it is almost impossible to make this happen.

expecting patch size: */switch*

After a **PC** switch, the compiler expects to find a decimal integer parameter. This error is reported if no such parameter is supplied.

more than one source file specified

The compiler will only compile one source file per run. Source file names on the command line are distinguished from switches by the fact that they do not start with the switch character. A common cause of this error is to type, for example:

```
t4c cats -fo dogs.bin
```

instead of:

```
t4c cats -fodogs.bin
```

There must not be a space before `dogs.bin` if it is to be regarded as a parameter of the `FO` switch.

range for patch size is 1 to 8 bytes

The message indicates that the `PC`*n* compilation option has been specified with an invalid value for the displacement value *'n'*. Refer to section 13.2.2 for a discussion of this option.

... *reason* ... **; please submit a CSR**

This message indicates a fault in the compiler itself. In some cases the *reason* may give a clue to a possible avoidance procedure but in all cases such messages should by reported to INMOS by means of a *Customer Software Report* (CSR).

If any other error messages have been generated before this fatal error message it is possible that a previous error has confused the compiler. Correcting the other errors may remove the cause of this message.

target must be /T4 or /T8 only: */switch*

This error is caused by using a switch like `-t9`, for example.

target processor already specified: */switch*

This could be caused by typing either of the following:

```
tc -t4 -t8 cats
t4c -t8 cats
```

or similar things. Each would flag the `T8` switch as an error; the first, because a `T4` switch has already been given, and the second because the `t4c` command implicitly specifies the T414 as the target processor.

unable to open *filename* **as listing file**

The host computer was unable to open the named file for output. This might be caused, for example, by using an erroneous filename:

```
t4c cats -fl99:zot
```

unable to open *filename* **as source file**

The given *filename* has been specified in the command which invoked the compiler but such a file cannot be accessed. Check that the filename has been spelled correctly and that it exists in the relevant directory.

unknown switch */switch*

The sequence of characters *-switch* was not recognised by the compiler as a valid switch.

Code generator errors

Once the syntactic and semantic phases of compilation have been completed the compiler attempts to generate transputer instructions for the program.

During this phase of compilation the compiler does not have access to the source program and so error messages cannot include the offending statement but simply give its line number.

The general form of these messages is:

Error: ...*reason* ... **at line** *number*

The following list describes the various *reason* messages, all of which are considered fatal error messages.

byte initialization too complex

This message is issued when the initialization of a byte-sized object (**char**) has specified a value which cannot fit into a byte. Note that in Parallel C the range of values held in a byte is [0, 255].

```
error()
{
   static char c = 1000;
}
```

count for shift must be in range [0..32]

This message is issued when the compiler tries to fold a constant expression and discovers that a shift count is not in the range [0,32].

```
error()
{
   int x;
   x = 7<<50;
}
```

initialization too complex

```
error()
{
   static int i;
   static int j = i;
}
```

Zero divide

This message is issued when the compiler tries to fold a constant expression and discovers that the divisor is zero.

```
error()
{
   int x,y;
   x = 1/0;
}
```

13.3.5 Errors in assembler code

This section deals with a number of special errors which may occur in assembler code. Other errors may occur in assembler code, and these are reported and dealt with in the usual way. The distinguishing mark of the errors dealt with in this section is that they are recognised at a relatively late point in the compilation. For this reason, although they are reported on the display, they are not output to the listing file. In order to save messages resulting from these errors, the output to the display could be redirected into a file.

These error messages have the following format:

opcode*:*message* **at line *ln* **in file** *fn*

opcode is replaced by the intruction mnemonic or pseudo-op coded on the line where the error happened. *ln* specifies the source line number. The file specification is omitted unless the error happened in an **#include** file, in which case *fn* is the filename in question.

In the descriptions below, only the *message* field is mentioned.

constant expected after "-"

This error would be reported for code like the following:

```
asm {
   ldc -foo;
}
```

Only a numeric constant is valid after the '–'.

operand form

This error will be reported when an opcode which cannot have a symbolic operand is given one. For example:

```
int foo;
asm {
   ldnlp foo;
}
```

`ldnlp` is not one of the opcodes allowed to have a symbolic operand.

constant expected

This error report occurs with the `byte` pseudo-op. For example:

```
int foo;
asm {
   byte foo;
}
```

The `byte` pseudo-op must be followed by a constant.

operand type wrong

This is reported when an opcode which is allowed to be followed by a symbolic operand is in fact followed by a label:

```
foo:
asm {
   ldl foo;
}
```

external operand not allowed

This error is reported when an opcode which is allowed to be followed by a symbolic operand is followed by an identifier with storage class **extern** which is not allocated storage by the declaration currently in scope.

```
extern int foo;
asm {
   ldl foo;
}
```

label required

This error may be reported for the **j** and **cj** opcodes. If these are followed by a symbolic operand, it must be an identifier defined as a label.

```
int foo;
asm {
   j foo
}
```

unknown opcode

This error is reported for assembler statements whose opcodes do not appear in the list in appendix C.

```
asm {
   foo 123;
}
```

syntax error

This error is reported when the format of the assembler statement is so peculiar that the compiler cannot understand it at all.

```
int foo;
asm {
   ldc 123 foo;
}
```

14 C language implementation

This chapter contains technical information about the way the C language is implemented on the transputer. Note that the information in this chapter applies only to the current version of the compiler; it is not guaranteed that future versions of the compiler will behave in the same way.

14.1 The C language

Currently, there is no internationally agreed standard for C. The definition of C adopted by 3L is the one given by Kernighan and Ritchie (the designers of the language) in *The C Programming Language*[1], except for the differences noted here. Henceforth their definition will be referred to as 'K&R C'. In order to use Parallel C, you will need access to the information in this book, which contains an excellent tutorial introduction to computer programming in general as well as the definition of C.

Because most other implementations of C are also based on Kernighan and Ritchie's definition it is possible to move C programs quite freely between different computers provided the programs avoid the use of extensions to K&R C peculiar to individual machines.

Unfortunately, although much of the power of C comes from the library functions for input and output of data, string handling and so on supplied along with most compilers, K&R C does not define a set of functions which all compilers must provide. This is not as much of a problem as it might be because most compilers agree about the definitions of the the commonly used functions. The library functions supplied with Parallel C (see chapter 15), with the exception of the *low-level* I/O functions, are common to almost all implementations of C; it is not guaranteed however that these functions will have exactly the same effect as their counterparts in other versions of C.

The differences between Parallel C and K&R C are described here. Section numbers in the text refer to the section numbers in the *C Reference Manual* (appendix A of *The C Programming Language*).

14.1.1 Restrictions

The following features of K&R C are not allowed in Parallel C.

Loose type checking of '.' and -> operators

Section 7.1 of K&R C states that the left operand of the operators '.' and -> must be a structure and the right must be the name of a member of that structure. Section 14.1 states however that the compiler allows any *lvalue* as the left operand of '.' and any expression of pointer or integer type as the left operand of ->. The rule of section 7.1 is followed, in disallowing examples like the following one:

```
int i;
struct { int p, q; } s;

i.p  = 0;
i->q = 17;
```

White space within compound operators

Assignment operators like '+=' are single tokens whose parts ('+' and '=') may not be separated by white space. If '+ =' is written instead of '+=', an error message will be printed by the compiler.

Use of `sizeof` in array declarations

Constant expressions used in an array declaration may not contain the **sizeof** operator. This example is illegal:

```
char v [ sizeof(int) ];
```

#line ignored

The **#line** compiler control line is accepted but ignored.

Anachronisms not allowed

Both of the anachronistic forms =op and `int x 3;` described in section 17 of K&R C are considered illegal.

Assignments should not be specified using the notation x=-1, rather the meaning should be made clear by use of one of the following forms:

```
   x -= 1;     /* meaning x = x - 1 */
or x = -1;     /* meaning x = (-1)  */
```

The correct modern form of the initialised declaration `int x 3;` is `int x = 3;`

14.1.2 Extensions

The following language features are allowed in Parallel C, but not in K&R C:

Dollar sign in identifiers

The dollar sign '$' may appear in identifiers. The dollar sign is treated as though it were a letter. The following are all acceptable identifiers:

```
$
rate$
$_max9
```

More significant characters in identifiers

Two identifiers are deemed by the compiler to be the same if their first 31 characters match (K&R C says 8 characters). Any additional characters are ignored. This rule also applies to external identifiers when a file of functions is being compiled.

If C programs are to be portable to many different compilers, they should only use identifiers which are distinct in the first 8 characters, except for external identifiers which should be distinct in the first 6 characters whether or not the distinction between upper and lower case letters is ignored.

Assignment to whole `struct/union` variables

In K&R C, all that can be done with a **`struct`** variable is to create a pointer to it (using the '**`&`**' operator) or access one of its members (using the '.' operator).

In Parallel C, the assignment operator '=' may be used to copy all of the members of a **`struct`** variable at once. If one operand of '=' is a **`struct`** then the other must be a **`struct`** of the same type. For example:

```
struct { int p, q; } x, y ;

x.p = 3;  x.q = 17;

y = x;    /* struct assignment */
```

After this structure assignment, **`y.p`** has the value 3 and **`y.q`** has the value 17.

```
struct tag { int p, q; } ;

clear(item)
struct tag item;
{
   item.p = 0;  item.q = 0;
}

example()
{
   struct tag pair;

   pair.p = 3;  pair.q = 4;
   clear(pair);
   return( pair.p + pair.q );
}
```

Figure 14.1 Example of the use of **struct** arguments

Both assignments in the example below are incorrect because the types of the operands for '=' do not match.

```
struct { int p, q; } x ;
struct { int a, b; } y ;
int i;

x = i;    /* one integer, one struct */
x = y;    /* same size, but different types */
```

Function arguments may also be **struct** types (K&R C allows only pointers to **structs** as arguments). **struct** arguments are declared and used in the same way as any other type.

The result returned by the function **example** in figure 14.1 will be 7 because, like all other types of function arguments in C, **struct** arguments are passed by value: **clear** cannot affect the contents of the structure **pair** which is passed to it, since it works with a copy of **pair** named **item**.

Restrictions on `struct` member names relaxed

In K&R C, the same member name may occur in different structures only if the fields identified by the member name and all preceding fields are the same. Parallel C makes no restrictions on the use of the same member name in different structures. Again, programs which must be usable with other C compilers should not make use of this fact.

type-name **syntax relaxed**

Kernighan and Ritchie give the definition of the *type-name* construct as:

type-name: type-specifier abstract-declarator

This allows only one *type-specifier* before the *abstract-declarator*, disallowing expressions like:

```
sizeof(long int)
(unsigned short) small
```

Multiple *type-specifiers* like `long int` are allowed in this context by other implementations of C, and by Parallel C.

14.1.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`int`	`extern`	`else`
`char`	`register`	`for`
`float`	`typedef`	`do`
`double`	`static`	`while`
`struct`	`goto`	`switch`
`union`	`return`	`case`
`long`	`sizeof`	`default`
`short`	`break`	`asm`
`unsigned`	`continue`	`fortran`
`auto`	`if`	`void`
`enum`		

The keywords `enum` and `void` are additional to the keywords defined in the *C Reference Manual* part of Kernighan and Ritchie's book. The keyword `entry` is not implemented.

14.2 System-dependent features

Using the features described in this section may cause different effects with different C compilers.

14.2.1 Data type enum not allowed

Some C compilers have extra data types which are not part of K&R C: **void** and **enum**. **void** is a special data type with no values, normally used to indicate that a function returns no value. The **enum** data type allows the programmer to construct new data types by enumerating the values which variables of that type may take, as in Pascal. For example, a variable of type **colour** might take any of the values **red**, **green** or **blue**.

Parallel C does not include **enum** types, but does provide **void**.

14.2.2 All bit fields unsigned

Parallel C only permits bit fields in structures to be integers. The class of integer (**int**, **short**, **long** etc.) is ignored: all bit fields are taken to be of type **unsigned int**. This restriction is permitted by K&R C (section 8.5).

14.2.3 >> operator

The use of the >> operator results in a logical shift rather than an arithmetic shift, that is, zeros are brought in at the most significant end of the operand rather than copies of the sign bit. As a result the value of the expression **(-1)>>1** is $7FFFFFFF_{16}$ and not $FFFFFFFF_{16}$ (−1).

14.2.4 Register variables

The **register** storage class is always ignored in Parallel C.

14.3 Predefined macros

The following macros are defined with the value '1' for every compilation:

_transputer _3L _IMST4 or **_IMST8** or **_IMST8A**

_IMST4 is defined for compilations done by **T4C** or **TC -T4**, **_IMST8** is defined for compilations done by **T8C** or **TC -T8**, and **_IMST8A** is defined for compilations done by **T8C -T8A** or **TC -T8A**. Any of these predefinitions may be cancelled by the **-U***mac* switch — see section 13.2.4.

14.4 Handling of #include files

Handling of **#include** lines is discussed in *The C Programming Language*[1] p. 207. When the compiler encounters an **#include** line, it searches for the specified file in a sequence of directories known as the *include list*. This consists of the following, which are searched in this order:

1 The current directory — except in the case of lines of this format:

#include *<filename>*

2 The 'standard places'. These are defined in one of the following ways:

- The user has defined the environmental variable **ISEARCH** to specify a series of directories.
- If the **X** compiler switch is used, the standard places are excluded from the include list.

3 Directories which have been specifically added to the include list at compilation time by means of the **I** switch — see section 13.2.3.

All the directories which are added to the include list, either by the **ISEARCH** environment variable or by the **I** compiler switch are assumed to be directories. If the filename specified in the **#include** line includes a directory specification, an attempt is made to concatenate it to each of the directories in the include list in order to find the file.

14.5 Assembly language

This section shows you how to use the 'in-line assembler' which is built into the C compiler to write programs containing embedded transputer assembly language instructions. It is assumed that you are already familiar with the transputer's architecture and machine code. If you are not familiar with these topics you will need to read in addition the *'Transputer instruction set: a compiler writer's guide'* [12].

If you use assembly language you may find the **decode** utility described in chapter 7 useful. It allows you to disassemble the object files generated by the compiler and read the machine code contained in them.

14.5.1 When to use assembly language

There are two main reasons for using in-line assembly language in a C program.

1 To take direct control of the hardware, for example to write a function which sets the transputer error flag.

2 To improve the performance of short sections of critical code.

The C compiler's in-line assembler is suitable for both these tasks. However, it is not intended for writing large sections of code in assembly language. If you need to do that, you should use a separate transputer assembler with its own macros, storage allocation directives and direct access to external symbols.

14.5.2 Assembly language syntax

Assembly language instructions are inserted into a program using the **asm** statement, which has the following syntax[1]:

*statement = "**asm**", "{ ", instructions, "} ";*
instructions = instruction, "; ", [instructions];

There are two basic forms of instruction, reflecting the division of the transputer instruction set into *direct* instructions which have an operand field, and the zero-address *indirect* instructions with no operand field[2] which take their operands from the three-register *evaluation stack.*

instruction = direct — indirect;
direct = direct opcode, operand;
indirect = indirect opcode;

Appendix C contains a list of the opcodes recognised by the compiler.

A function to set the transputer error flag could be written as:

```
set_error_flag()
{
   asm { seterr; }
}
```

Two more example **asm** statements are shown below.

```
asm { seterr; stopp; }
asm { ldl 0; ajw -10; stl 2; ldc 123; stl 1; }
```

[1]See section 17.1, Standard Syntactic Metalanguage

[2]Actually, there are only direct operations. All the indirect operations are assembled as particular literal operand values for one direct instruction called **opr**.

14.5.3 Literal operands

The operand of a direct instruction can be any literal 32-bit integer value. The assembler automatically generates any **pfix** or **nfix** bytes required to encode large values.

operand = constant;

Decimal, octal and hexadecimal constants can be used; floating-point, character and string constants are not allowed. Some valid examples are shown below.

```
#define XYZ 23
asm {
   ldc   17;    /* decimal */
   ldc   0xff;  /* hex */
   ldc   0377;  /* octal */
   ldc   XYZ;   /* decimal 23, defined by macro */
}
```

Note that constant expressions like **sizeof(int)** or **10+7** are not allowed as assembly language operands.

14.5.4 Variables as operands

The assembler allows C variables to be used as operands for the following direct instructions:

ldl which loads a data word from memory and pushes it onto the evaluation stack;

stl which pops a word from the evaluation stack and stores it in memory;

ldlp which pushes a pointer to a word in memory onto the evaluation stack.

The required syntax is:

*instruction = "***ldl***", identifier —*
*"***ldlp***", identifier —*
*"***stl***", identifier;*

We can now write a complete C example function which uses assembly language to manipulate program variables.

```
main()
{
   int a, b=123, c=456;
   asm {
      ldl b;  ldl c;        /* load b and c */
      add;                  /* add them */
      stl a;                /* store result in a */
   }
   printf("a=%d\n", a);
}
```

Storage class

An identifier used as the operand of a **ldl**, **ldlp** or **stl** instruction must be the identifier of a variable. The variable can have storage class **auto**, **register** or **static**[34]. An **extern** variable can also be used, but only in the scope of the declaration which actually allocates storage for the variable. The following example is allowed:

```
int i = 17;  /* storage for 'i' allocated here */
fun()
{
   asm { ldc 123; stl i; }
}
```

The next example is not allowed, because storage for **j** is not allocated by the declaration in scope. That declaration contains an explicit **extern** keyword, which means that storage for **j** is allocated elsewhere (probably in a different file).

```
extern int j;  /* refers to storage elsewhere */
fun2()
{
   asm { ldc 123; stl j; }
}
```

[3]The assembler automatically generates the extra **ldl** instruction required to load the base address of the static area and converts the 'local' operation into a 'non-local' one.

[4]Restriction in V2.0: if a **static** or **extern** variable is accessed from within an **asm** statement, there must be at least one C statement in the enclosing function which also uses a **static** or **extern** variable (not necessarily the same one).

Type

Identifiers used as operands for **ldl**, **ldlp** and **stl** must be declared as variables. Function identifiers, labels, **struct** member names, and tags like **struct** tags cannot be used.

Otherwise the type of a variable is ignored when it is used as an assembly language operand. The **ldl** and **stl** instructions always load or store exactly one word, irrespective of how a variable was declared. If an object (e.g. a **struct**) is longer than a word then only the first word is accessed. Take care with **char** objects, which are shorter than a word: the whole word beginning at the address of the **char** will be loaded or stored to.

14.5.5 Accessing complex structures

Expressions are not allowed as assembly language operands. The following example shows some incorrect operands.

```
struct s { int value; struct s *link; };
int total=0;

sum(p)
struct s *p;
{
   while (p) asm {
      ldl   p->value;    /* wrong: p->value is */
                         /*       an expression */
      ldl   total;       /* ok */
      add;
      stl   total;
      ldl   p;
      ldnl  link;        /* wrong again: link */
                         /*  is a member name */
      stl   p;
   }
}
```

To make this example work, we can rewrite it as follows.

```
struct s { int value; struct s *link; };
int total=0;

sum2(p)
struct s *p;
{
   while (p) asm {
      ldl   p;  /* load pointer */
                /*to base of struct */
      ldnl  0;  /* value: 0 offset */
                /* from struct base */
      ldl   total;
      add;
      stl   total;
      ldl   p;  /* struct base addr again */
      ldnl  1;  /* link: offset=1 word */
      stl   p;
   }
}
```

In general, an object whose address is given by a complex expression (e.g. an array element) can be manipulated in assembly language either by saving a pointer to the object beforehand in C and then accessing the object via the pointer, or by working out how the compiler will allocate storage for the object and then calculating its address in assembly language.

For example, to store the character '*' in element `i` of a `char` array `A` we can use any of the following techniques.

1 Write in C.

```
char A[128];
int i;
f1() { A[i] = '*'; }
```

2 Save a pointer to the object in C.

```
f2() {
   char *p = &A[i]; /* save pointer to it */
   asm {
      ldc   0x2A;    /* ASCII '*' */
      ldl   p;
      sb;            /* store byte */
   }
}
```

3 Calculate the object's address in assembly language.

```
f3() {
   asm {
      ldc   0x2A;
      ldl   i;
      ldlp  A;
      bsub;       /* byte subscript: &A[i] */
      sb;
   }
}
```

Use methods 1 or 2 if at all possible. If you use method 3 you may find that your program will not work with future versions of the compiler because the way in which storage is allocated for some object changes. If you do need to use method 3, the **decode** utility described in chapter 7 can be used to find out how the compiler has allocated storage for a program's variables.

14.5.6 Labels and jumps

In the examples given so far, C control statements (e.g. **while**) have been used to control the execution of assembly language statements. Sometimes though, you may need to program jumps in assembly language. For example, you might want to avoid storing an intermediate result back into a local variable in order to be able to test its value using a C conditional statement.

To make programming jumps easy, the **j** and **cj** instructions permit C labels to be used as operands.

instruction = "j", label —
"cj", label; label = identifier;

An identifier used as the operand of a **j** or **cj** instruction must appear as a C statement label somewhere in the body of the enclosing function.

The example below shows the list-summing function with its **while** statement recoded in assembly language.

```
struct s { int value; struct s *link; };
int total=0;

sum3(p)
struct s *p;
{
      /* access extern variable in C before
         using it in asm */
      total = 0;
loop: asm {
      ldl   p;
      cj    out;
      ldl   p;
      ldnl  0;     /* p->value */
      ldl   total;
      add;
      stl   total;
      ldl   p;
      ldnl  1;     /* p->link */
      stl   p;     /* p = p->link; */
      j     loop;
   }
out:
}
```

Note the forward reference to label **out**. Any identifier which appears in an **asm** statement and which has not yet been declared is automatically declared as a forward reference to a label, which must be defined before the end of the function.

Labels within asm statements

C labels must be attached to C statements. It is not possible to label individual instructions within an **asm** statement. If you need to do so, the instruction sequence must be split up into multiple **asm** statements, each of which can be labelled.

```
asm {
   ldc   17;
L: stl   i;
}
```

The above example is incorrect, because a label has been put inside an **asm**

statement. It must be split up:

```
    asm { ldc 17; }
L:  asm { stl i; }
```

Jump optimisations

The assembler always generates minimum sized jumps. Note that it may also delete unreachable jumps and merge jumps-to-jumps.

14.5.7 Literal machine code

The assembler allows you to put literal machine code directly into the object file using the **byte** pseudo-operation.

instruction = "**byte**", *code list*; *code list* = *constant*, { ",", *constant* };

For example, the following **asm** statement outputs the actual machine code for a **ret** instruction:

```
asm { byte 0x22, 0xF0; }
```

14.5.8 Errors

The messages produced by the compiler when it detects an error in an assembly language statement are of the form:

*****_opcode: message_ **at line** _n_ **of file** _f_

The opcode, line and file parts refer to the name and location of the offending instruction; the various possible messages are included in the full list of compiler error messages in chapter 14. The filename part of the message is omitted unless the error is within an **#include** file.

The following error message can appear if you mis-spell an identifier in an **asm** statement:

label "*ident***" is used in function "***f***" above but is not defined there**

This is because the mis-spelt identifier is assumed by the compiler to be a forward reference to a label.

14.6 Data-type representations

The primitive C data types are represented on the transputer as follows:

char	byte	(unsigned)
int	word	
short	word	
long	word	
float	word	(IEEE single-precision format)
double	2 words	(IEEE double-precision format)
pointer	word	

On T414 and T800 transputers, a byte is 8 bits and a word is 32 bits.

char variables occupy 8 bits, and can hold values between 0 and 255.

unsigned short variables occupy 32 bits, but only the 16 least significant bits are used in expressions.

The IEEE floating-point formats used to hold **float** and **double** quantities on the transputer are described in detail in the IEEE floating-point standard[9]. The way in which these standard formats are represented in transputer memory is shown in figure 14.2.

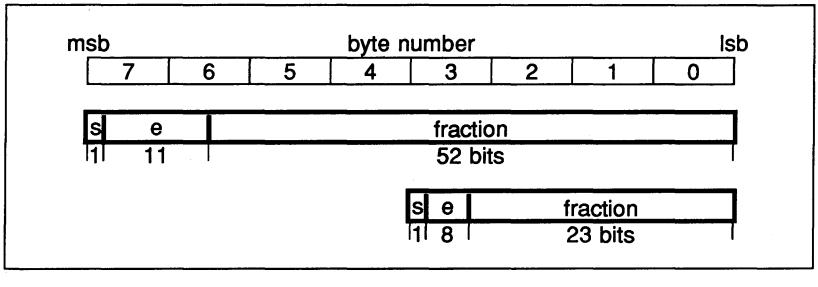


Figure 14.2 Representation of floating-point values

All types except **char** are automatically word aligned by the compiler.

struct and **union** types are always rounded up to a whole number of words, even if they contain only byte objects.

Successive bit fields in a **struct** are allocated starting from the least significant (lowest addressed) end of a word. Only integer fields are allowed, and plain **int** fields are treated as **unsigned**. No field may be wider than 32 bits.

```
struct s1 { char first;
            int bits1:7, bits2:7;
            char last; };

struct s2 { char first, last;
            int bits1:7, bits2:7; };

struct s3 { char first;
            int bits1:7;
            char last;
            int bits2:7; };
```

Figure 14.3 Effect of Alignment on **struct** Size

Adjacent bit fields are considered together when they are being packed into words. A sequence of fields occupying up to 8 bits is packed into the next byte in the structure. Longer sequences are aligned starting at the next word in the structure and padded out to a whole number of words (even if following **char** fields could otherwise be packed into this padding space). Figure 14.3 shows the effects of this on the total size of structures. In **s1** the fields **bits1** and **bits2** together occupy 14 bits and are therefore aligned to start at the next word boundary (offset 4 bytes). They occupy the whole of this word, forcing **last** into the next word (offset 8 bytes), making **s1** 3 words long after being rounded up from 9 bytes.

In **s2** the two char items **first** and **last** have been brought together reducing the size of the struct to two words.

In **s3** the bit fields have been separated by **last**. This prevents the bit fields being combined into a unit of 14 bits, leaving them as two byte-sized objects. The overall effect is to reduce the size of the struct to four bytes (1 word).

15 The C run-time library

15.1 Purpose of the run-time library

The Parallel C run-time library is a collection of compiled functions which perform commonly-used operations not included in the C language itself: reading and writing data, and evaluation of mathematical functions like **sin** and **cos** are the most obvious instances.

This chapter describes the conventions used to call library functions and describes their arguments and lists the available routines grouped by function (I/O, string handling, etc.). Chapter 16 lists the available routines in alphabetic order, giving a description of the effects of each.

15.2 Conventions

This section describes how to use standard header files in calling library functions and how to interpret the notation used in chapter 16 to specify the number and types of arguments they require.

Run-time library functions are used in exactly the same way as user-defined functions (most are in fact just normal C functions anyway). To use a library function, a program must first declare the name of the function to be used, and indicate that it is external to the program (storage class **extern**).

So that the declarations of library functions in user programs are always correct, standardised *header files* are provided with the system for each group of library functions. The programmer uses the C **#include** statement to access the contents of the header file before making use of any of the functions declared there. As well as containing the required function declarations, the header file will include declarations for any special data types required by its functions. For example, consider the standard I/O functions. These are declared in the header file **stdio.h**. Before the first use of any of the standard I/O functions, a program must contain the statement:

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

This declares all of the standard I/O functions like **printf** and **getc** as well as defining the macros **EOF** and **NULL** which are used in communication between the I/O functions and user programs. **EOF** has the value (−1); **NULL** has the value 0.

Programs should always use the header files provided with the compiler rather than attempting to provide their own declarations for library functions since the

declarations of some functions will differ from the obvious declaration implied by the function synopses in this chapter.

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 header files, if any, must be used in order to access the function. For example, the synopsis for the **fgets** function looks like this:

```
#include <stdio.h>
char *fgets(str, n, stream)
char *str;
int n;
FILE *stream;
```

This means that **fgets** returns a result of type **(char *)** and has three arguments of types **(char *)**, **(int)** and **(FILE *)**, where **FILE** is a data type declared in the header file **stdio.h**. This header file must be included in all programs which use the function.

Ellipsis is used in function synopses to indicate that a function has a variable number of arguments, for example the **printf** function:

```
#include <stdio.h>
printf(format[,arg1[,arg2[,...]]])
char *format;
```

The synopsis shows that **printf**'s first argument must be a character pointer. The square brackets **[]** indicate that the enclosed arguments are optional; ellipsis '...' indicates repetition. Where argument types are not shown in the synopsis (e.g. **arg1**, **arg2**, ... for **printf**) the allowed argument types are discussed in the text.

Functions implemented as macros are marked with a dagger (†).

15.3 Header files

The following header files are supplied with the compiler.

ascii.h	**errno.h**	**sema.h**	**thread.h**
assert.h	**limits.h**	**serv.h**	**time.h**
boot.h	**malloc.h**	**setjmp.h**	**timer.h**
chan.h	**math.h**	**stdio.h**	
chanio.h	**net.h**	**stdlib.h**	
ctype.h	**par.h**	**string.h**	

15.4 Library modules

This section lists the library functions provided, divided into the following functional groups.

- Stream I/O to files and devices, including facilities for random file access
- Classification of ASCII characters (e.g. is a character an ASCII letter?)
- String manipulation, including string copy and string comparison
- Character conversion (e.g. convert uppercase letters to lowercase)
- Numeric conversions between ASCII string and binary representations for integer and floating-point values
- Mathematical functions, including logarithms and trigonometric functions
- Dynamic ('heap') memory allocation and deallocation
- Creating additional execution 'threads'
- Operations on software semaphores
- Accessing the transputer's timer facilities
- Accessing the transputer's message-passing primitives
- Support for the flood-fill configurer's network protocol
- Functions to allow multi-threaded access to the run-time library
- Various other miscellaneous functions

Some background information common to all of the functions in a group is presented in this section rather than being repeated with the description of each individual function. In particular, the concepts on which the standard I/O system is based are presented here: *stream, file pointer* etc.

15.4.1 Input/output

The functions which are provided to read and write data fall into two groups: the *low-level* I/O functions and the *standard* I/O functions.

Standard I/O

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 byte at a time). On output, user data is placed in a data 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 an overall 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 file **`stdio.h`** before they may be called.

Before a user of the standard I/O package can read or write the data in a file, a path to the file must be *opened* by calling the **`fopen`** function. The name of the file is passed to **`fopen`**, which, if the file is accessible, returns a pointer to a structure of type **`FILE`**. This *file pointer* must be used by the calling program 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 data type **`FILE`** is declared in the header file **`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 which may be open at once. Files may be opened again after they have been closed. Having more than one path 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 the end of a program is, however, unnecessary; this is done automatically by the standard I/O system.

```
#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 */
}
```

Figure 15.1 An example of using **fopen** and **fclose**

Figure 15.1 gives an example where a file named **fred** is opened, some ASCII data is written out to it and the file is closed. For clarity, no error checking is performed.

For convenience, three file pointers are always automatically opened. These are declared in **stdio.h** as follows.

FILE *stdin; This is the standard input stream. By default on most systems, **stdin** is connected to a terminal keyboard.

FILE *stdout; This is the standard output stream. **stdout** on most systems is the display device (VDU or printer) of a terminal.

FILE *stderr; This is the standard error stream, used by programs for outputting error messages. It too is normally opened on the terminal output device.

To simplify writing programs which 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 command language) 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 terminal for standard input and output, then run unchanged using files for input and output, yet the program itself need not open files.

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. Random file access is supported by allowing user positioning of the file pointer.

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 terminals 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.

Binary I/O

The basic units in the above discussion of stream I/O are 'characters': values of type **char**. These are integers which stand for graphic character representations in the encoding scheme of the host computer system (e.g. the ASCII encoding for 'A' is 65, in the EBCDIC scheme used by IBM it is 193). The C I/O system, however, does not require that the values transferred be valid character representations. In fact, any binary value which can be represented in a **char** variable may be written to a file (and later read back unaltered). In Parallel C any value in the range 0 to 255 will fit in a **char**. Arbitrary binary data can be stored in files using the standard I/O system by recording it as sequences of **char** values.

By default C reads and writes text files. If you need to process binary data you must inform the run-time library that a particular file is to be processed as a binary file. This can be done either by using the 'binary' specifier **b** in a call to **fopen** (for example, **fopen("x.bin", "rb")**) or by altering the setting of the global variable **extern int _fmode;**, which controls the run-time library's default behaviour.

You can assign one of two flag values to **_fmode**:

- **O_TEXT**
- **O_BINARY**

Both of these flag values are defined as macros in the **<stdio.h>** header file.

For example, to open a binary file **binfile** for input, you might write:

```
#include <stdio.h>

main()
{
   FILE *fp;
   extern int _fmode;      /* declare _fmode */

   _fmode = O_BINARY;      /* open for binary access */

   fp = fopen("binfile", "r");
}
```

Note that **_fmode** will not be altered by the C library: if you wish to open a text file again once you have set **_fmode** to **O_BINARY**, you must explicitly change **_fmode** back to the default value of **O_TEXT**: **_fmode** = **O_TEXT**.

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.

Text I/O

Text I/O in C is simply a special case of the binary I/O discussed above where the values transferred are restricted to the valid character codes for the host system.

Human-readable text files are divided into lines. Line-breaks are represented in the stream I/O model by the *newline* character, '**\n**'. The integer code for this character is system-dependent. On output, newline characters may be included at arbitrary points in the text. On input, programs detect the end of a line by comparing characters being read with the value '**\n**'.

Standard I/O functions

The library functions which form the standard I/O package are listed in this section.

`clearerr`	resets the error and end of file indicators
`fclose`	closes a file
`fdopen`	creates a `FILE` structure and associates it with a file descriptor
`feof`†	tests for end-of-file
`ferror`†	returns a nonzero integer if an error occurs during read or write operations
`fflush`	writes out any buffered information to the file
`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
`fileno`†	returns an integer file descriptor
`fopen`	opens a file
`fprintf`	performs formatted output to a specified file
`fputc`	writes a single character to a file; generates a true function call
`fputs`	writes a string to a file
`fread`	reads a specified number of items from the file
`freopen`	reassigns the address of a `FILE` structure and reopens the file
`fscanf`	performs formatted input from a file
`fseek`	places the file pointer at a specified byte offset relative to the beginning of the file, the end of the file or the current location in the file
`ftell`	returns the current byte offset from the beginning of the file to the current location within the file
`fwrite`	writes the specified number of items to a file
`getc`†	returns the next character from a file; implemented as a macro
`getchar`†	returns the next character from the standard input device
`gets`	reads a line from the standard input device; the newline is replaced with a NUL character
`printf`	performs formatted output to the standard output device
`putc`†	writes a single character to a file; implemented as a macro
`putchar`†	writes a single character to the standard output device
`puts`	writes a string to the standard output device; terminates the string with a newline
`putw`	writes a specified integer to a file

`rewind`	places you at the beginning of the file
`scanf`	performs formatted input from the standard input device
`setbuf`	associates a buffer with an input or output file
`sscanf`	performs formatted input from memory
`sprintf`	performs formatted output to a character string in memory
`ungetc`	writes a character to a file buffer and leaves the file positioned before the character

Low-level I/O

The low-level I/O functions transfer 'raw' user data to or from files or devices in variable length blocks (down to one byte). The low-level I/O functions are provided mainly for compatibility with other implementations of C; normally standard I/O should be used. In low-level I/O files are accessed via 'file descriptors', small integers returned by the system when a file is opened. Other functions are provided to create new files and directly control the position in a file where data transfers will take place.

The low-level I/O functions are:

`close`	closes a file
`creat`	creates a new file
`isatty`	determines if a file descriptor is associated with a terminal
`lseek`	places you at a byte offset within a file and returns the new position as an integer
`open`	opens a file for reading, writing or both
`read`	reads a specified number of bytes from a file and places them in a buffer
`write`	writes a number of bytes from a buffer to a file

15.4.2 Mathematical functions

The mathematical functions calculate various standard mathematical functions such as logarithms, sines, cosines etc.

The trigonometric functions operate on angles expressed in radians.

Errors are handled by returning impossible or unusual result values and setting an error code in the external integer variable `errno`.

abs	returns the absolute value of the integer argument
acos	returns the arc cosine of the argument
asin	returns the arc sine of the argument
atan	returns the arc tangent of the radian argument
atan2	returns the arc tangent of the division of the arguments
ceil	returns the smallest value which is equal to or greater than the argument
cos	returns the cosine of the radian argument
cosh	returns the hyperbolic cosine of the argument
exp	returns the base e raised to the power of 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
frexp	split 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
pow	returns the value of the first argument raised to the power of the second argument
sin	returns a value that is the sine of the radian argument
sinh	returns a value that is the hyperbolic sine of the argument
sqrt	returns the square root of the argument
tan	returns the tangent of the argument
tanh	returns the hyperbolic tangent of the argument

15.4.3 String handling

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.

index	find character in string
memcpy	copies a given number of bytes from one memory location to another
memset	overwrites each byte of an object with a given character code
rindex	find character in string

`strcat`	concatenates two strings
`strchr`	finds a specified character in a string
`strcmp`	performs lexicographic comparison of two ASCII strings
`strcpy`	copies one string to another
`strcspn`	returns the length of the initial part of a string which does not contain specified characters.
`strlen`	returns the length of a string
`strncat`	concatenates two strings up to a maximum number of characters
`strncmp`	performs lexicographic comparison of two ASCII strings (up to a maximum number of characters)
`strncpy`	copies a maximum number of characters from one string to another
`strspn`	returns the length of the initial part of a string which contains specified characters.
`strtok`	returns a pointer to the first character of a token.

15.4.4 Character classification

The character classification functions described here are implemented as macros. They return a nonzero 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
`isascii`†	determines if the argument is an ASCII character
`iscntrl`†	determines if the argument is an ASCII control character
`isdigit`†	determines if the argument is a digit
`isgraph`†	determines if the argument is a printing character but not a space
`islower`†	determines if the argument is a lowercase 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 uppercase letter
`isxdigit`†	determines if the argument is a hexadecimal digit character

15.4.5 Conversions

These functions provide conversion operations between various representations of numeric values: binary integers, binary floating-point and character string. Some character mapping functions (upper/lower case mapping) are also provided.

atof	converts an ASCII string to a numeric value (**double**)
atoi	converts an ASCII string to a numeric value (**int**)
atol	converts an ASCII string to a numeric value (**long**)
tolower	converts uppercase characters to lowercase; returns lowercase characters unchanged
_tolower†	converts uppercase characters to lowercase; returns lowercase characters unchanged
toupper	converts lowercase characters to uppercase; returns uppercase characters unchanged
_toupper†	converts lowercase characters to uppercase; returns uppercase characters unchanged

15.4.6 Dynamic memory allocation

Building complex dynamically changing data structures requires a different class of storage from **static** or **extern** variables (which must be preallocated by the programmer when a program is written and are therefore not flexible enough) and **auto** or **register** variables (which disappear when the procedure which created them returns; some dynamic data structures must be operated on by many procedures).

This extra storage class is generally referred to as *heap* storage. In C, heap storage is allocated by calling a library function (**malloc**) and remains until it is explicitly released by calling another function (**free**).

calloc	allocates and clears an area of memory
cfree	deallocates the space allocated by **calloc** or **realloc**
free	deallocates the space allocated by **malloc** or **realloc**
malloc	allocates the specified number of contiguous bytes of memory
realloc	changes the size of an area previously allocated by **malloc** or **realloc**

15.4.7 Date and time

The following functions return information about the time.

`clock`	returns processor time used
`time`	returns the current calendar time

15.4.8 `thread` package

The functions in this section allow a Parallel C program to create new threads of execution within a single task.

Every thread executing on a transputer has a priority, which is either 'urgent' or 'not urgent' The header file **<thread.h>** defines the following literals to represent this:

- **THREAD_URGENT**
- **THREAD_NOTURG**

`thread_start`	general thread-starting facility
`thread_create`	simpler shorthand version of `thread_start`
`thread_priority`	returns current thread's priority
`thread_deschedule`	make current thread momentarily unable to execute
`thread_restart`	restart a thread given a workspace pointer
`thread_stop`	stop the current thread

15.4.9 `sema` package

This group of functions allows a Parallel C 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 new type **SEMA** which is used by these functions.

`sema_init`	initialise a semaphore
`sema_signal`	perform the *signal* operation on a semaphore
`sema_wait`	perform the *wait* operation on a semaphore
`sema_signal_n`	perform `sema_signal` n times
`sema_wait_n`	perform `sema_wait` n times

15.4.10 `timer` package

Each transputer associates a hardware timer with the group of threads executing at a particular priority. The following functions allow threads to manipulate the timer associated with the priority at which they are executing.

`timer_after`	indicates whether one time value is later than another
`timer_delay`	wait at least a specified number of ticks
`timer_now`	returns the current timer value
`timer_wait`	wait until current timer reaches some value

15.4.11 `chan` package

The functions described here allow programs to access the transputer's basic communication facility, which is to transfer a *message* across a *channel*. The header file **<chan.h>** defines the following:

- a type **CHAN** representing the channel data type
- **(CHAN *)** literals for the input and output channels for each of the four INMOS links attached to the transputer
- a **(CHAN *)** literal for the channel associated with the transputer's external event mechanism
- a **CHAN** literal for initialising channels to their inactive state
- procedures to initialise and reset channels
- procedures to send and receive communications across channels, with variants to wait until the communication occurs or to fail after some time-out interval.

The literals defined by **<chan.h>** are as follows; note that these literals are not entered in the alphabetical list of library entry points.

`Link0Input`	input channel associated with link 0
`Link0Output`	output channel associated with link 0
`Link1Input`	input channel associated with link 1
`Link1Output`	output channel associated with link 1
`Link2Input`	input channel associated with link 2
`Link2Output`	output channel associated with link 2
`Link3Input`	input channel associated with link 3
`Link3Output`	output channel associated with link 3
`EventReq`	channel associated with external events
`NotProcess_P`	value to which channel words are initialised

The functions provided in the 'chan' package are as follows:

`chan_init`	initialise a channel word
`chan_reset`	resets a channel, along with any link hardware associated with it.
`chan_in_byte`	input a byte from a channel
`chan_in_byte_t`	as above, with timeout
`chan_in_word`	input a word from a channel
`chan_in_word_t`	as above, with timeout
`chan_in_message`	input a message from a channel
`chan_in_message_t`	as above, with timeout
`chan_out_byte`	output a byte to a channel
`chan_out_byte_t`	as above, with timeout
`chan_out_word`	output a word to a channel
`chan_out_word_t`	as above, with timeout
`chan_out_message`	output a message to a channel
`chan_out_message_t`	as above, with timeout

15.4.12 `net` package

The functions described here allow tasks running under the flood-filling configurer's network protocol to communicate without knowing the exact details of that protocol.

`net_send`	send a message into the network
`net_receive`	receive a message from the network

15.4.13 `par` package

In a program in which many execution threads are active, access to the C run-time library must be synchronised, so that only one thread may be performing a library operation at one time. For example, if two threads attempted to allocate a memory block at the same time (say, using `malloc`) then the run-time library's data structures could become corrupted. The required synchronisation is achieved by a `SEMA` variable `par_sema` defined in the header file `<par.h>`, which should be used by any thread wishing to use the C run-time library and released when it is finished.

As an alternative, some of the more common functions used in concurrently executing threads are available in an interlocked form, which include these semaphore operations.

par_malloc	interlocked version of **malloc**
par_free	interlocked version of **free**
par_printf	interlocked version of **printf**
par_fprintf	interlocked version of **fprintf**

15.4.14 Compatibility channel I/O

The functions in this category provide compatibility with previous versions of the run-time library. To perform I/O operations on channels, new programs should use the functions described in section 15.4.11.

_outword	output a word to a channel (use **chan_out_word**)
_outbyte	output a byte to a channel (use **chan_out_byte**)
_outmess	output a message to a channel (use **chan_out_message**)
_inmess	input a message from a channel (use **chan_in_message**)

15.4.15 Miscellaneous

Other useful library functions are provided for halting program execution, non local jumps, debugging etc.

assert	program debugging routine
boot_peek	peek at memory of (unbooted) neighbouring transputer
boot_poke	poke into memory of (unbooted) neighbouring transputer
exit	stop program
getenv	access environment variables
longjmp	returns to the context saved by **setjmp**
rand	pseudo-random number generator
remove	removes a file from the file system
serv_filter	generates INMOS host file server protocol filter threads
setjmp	saves the context of the calling function for a subsequent **longjmp** call
strtol	convert string to long integer
strtoul	convert string to unsigned long integer
srand	change seed for **rand**
system	execute operating system command string
unlink	removes a file from the file system

15.5 The C main program

The C **main** program function is called with the following parameters.

```
typedef int CHAN;

main(argc, argv, envp, in, inlen, out, outlen)
int argc, inlen, outlen;
char *argv[], *envp[];
CHAN *in[], *out[];
```

argv[0] is the program name, currently always a pointer to a null string (i.e. a pointer to a '\0' character).

If the value of **argc** is greater than one then **argv[1]...argv[argc-1]** are pointers to token strings each of which is terminated by '\0'.

argv[argc] is a null pointer.

argc is the number of tokens, including the program name. It is always greater than zero.

envp is always **NULL**.

in and **out** are vectors of pointers to channels. **inlen** and **outlen** are the number of elements in **in** and **out** respectively.

The C program can send and receive messages across these channels using the channel I/O functions described in section 15.4.11.

15.6 Reduced run-time library

If the only run-time library functions a C program needs are the channel I/O functions, the size of the resulting executable program can be reduced and the supporting filer process can be dispensed with by using the *reduced* run-time library instead of the normal C run-time library.

The reduced library contains only essential initialisation code and the channel I/O functions. A program linked with the reduced library cannot use standard I/O functions like **printf**, or utility functions like **exit**.

The user-written function **main** is called by the reduced library with exactly the same arguments as shown in section 15.5 above, but **argc** is always 1, **argv[0]** is always "", and **argv[1]** is always **NULL**. That is, the command line arguments are not passed in to the program.

The T414 and T800 versions of the reduced run-time library are supplied in the file **sacrtl.lib**.

Programs which are to be run on a remote node in a transputer network (using the configurers **config** and **fconfig**) must be linked with both the reduced run-time library and the appropriate harness module (**taskharn.t4x** or **taskharn.t8x**), for example:

```
t4c x
ilink taskharn.t4x x.bin sacrtl.lib -o x.c4x
iboot x.c4x -c -o x.b4
```

16 Alphabetic list of run-time library functions

This chapter lists all of the supported library functions supplied with Parallel C. The functions are arranged in alphabetical order; note that non-letters such as digits or '_' are regarded as being 'before' the alphabet. Thus, a function **a_a** would appear before **aaa**, and functions whose names begin with '_' appear at the start of the list.

_inmess read message from channel

```
#include <chanio.h>
_inmess(chanp, buf, nbytes)
int *chanp, nbytes;
char buf[];
```

Reads a message of length **nbytes** from the channel pointed to by **chanp** into the buffer **buf**.

This function is provided only for compatibility with older versions of the run-time library. New programs should use the equivalent function **chan_in_message**.

_outbyte write byte to channel

```
#include <chanio.h>
_outbyte(b, chanp)
char b;
int *chanp;
```

Writes a single-byte message consisting of the value **b** to the channel pointed to by **chanp**.

This function is provided only for compatibility with older versions of the run-time library. New programs should use the equivalent function **chan_out_byte**.

_outmess write message to channel

```
#include <chanio.h>
_outmess(chanp, buf, nbytes)
int *chanp, nbytes;
char buf[];
```

Writes a message of length **nbytes** from the buffer **buf** to the channel pointed to by **chanp**.

This function is provided only for compatibility with older versions of the run-time library. New programs should use the equivalent function **chan_out_message**.

_outword write word to channel

```
#include <chanio.h>
_outbyte(w, chanp)
int w;
int *chanp;
```

Writes a four-byte message consisting of the value **w** to the channel pointed to by **chanp**.

This function is provided only for compatibility with older versions of the run-time library. New programs should use the equivalent function **chan_out_word**.

_tolower† convert char to lower case

```
#include <ctype.h>
int _tolower(cval)
int cval;
```

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.

_tolower behaves like **tolower** but is implemented as a macro.

_toupper† convert char to upper case

```
#include <ctype.h>
int _toupper(cval)
int cval;
```

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.

_toupper behaves like **toupper** but is implemented as a macro.

abs integer absolute value

```
#include <math.h>
int abs(arg)
int arg;
```

abs returns the absolute value of its integer operand. The result returned by **abs** is implementation defined if **arg** is the largest negative integer.

acos calculates the arc cosine of its argument

```
#include <math.h>
double acos(x)
double x;
```

acos returns the arc cosine in the range $[0, \pi]$.

asin calculates the arc sine of its argument

```
#include <math.h>
double asin(x)
double x;
```

asin returns the arc sine in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

assert program debugging routine

```
#include <assert.h>
assert(expression)
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.

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 the message 'assertion failed' on the standard error file and performs a diagnostic traceback of the call stack.

No value is returned by **assert**.

atan arc tangent

```
#include <math.h>
double atan(x)
double x;
```

atan returns the arc tangent of **x**.

atan2 arc tangent of the division of its arguments

```
#include <math.h>
double atan2(x, y)
double x, y;
```

atan2 returns the arc tangent of $\frac{x}{y}$ in the range $[-\pi, \pi]$.

atof convert string to floating point

```
double atof(nptr)
char *nptr;
```

The string pointed to by **nptr** is converted to double-precision floating point representation. The first unrecognised character terminates the string.

atof recognises an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

The effect of overflow is implementation defined.

atoi convert string to integer

```
int atoi(nptr)
char *nptr;
```

This function converts the string pointed to by **nptr** to integer representation. The first unrecognised character ends the string.

atoi recognises an optional string of tabs and spaces, then an optional sign, then a string of digits.

The effect of overflow is implementation defined.

atol convert string to long integer

```
long atol(nptr)
char *nptr;
```

This function converts the string pointed to by **nptr** to long integer representation. The first unrecognised character ends the string.

atol recognises an optional string of tabs and spaces, then an optional sign, then a string of digits.

In Parallel C, **atol** is equivalent to **atoi** since **sizeof(int)** and **sizeof(long int)** are the same.

The effect of overflow is implementation defined.

boot_peek peek in memory of neighbouring transputer

```
#include <boot.h>
int boot_peek(ad, val, chan_in, chan_out)
int ad, *val;
CHAN *chan_in, *chan_out;
```

This function reads a word of memory from address **ad** in a neighbouring transputer into the variable pointed to by **val**. In order to be able to do this, the neighbour transputer must have been recently reset but *not* bootstrapped. In this special state, the transputer processor executes special firmware implementing a 'peek and poke' protocol described in the INMOS data sheets[10,11] and the *Transputer instruction set: a compiler writer's guide*[12]. The function returns a non-zero value if the 'peek' operation succeeds.

The neighbouring transputer is connected to the one on which the **boot_peek** function is executed by an INMOS link, with which are associated an input and output channel **chan_in** and **chan_out**. If that link does not lead to another transputer, or if the other transputer is not executing the 'peek and poke' firmware, the **boot_peek** function will time out after 30 ticks of the transputer timer associated with the current thread's priority. This timeout period is around 2mS for a non-urgent thread. If **boot_peek** times out, it returns zero.

boot_poke poke to memory of neighbouring transputer

```
#include <boot.h>
int boot_poke(ad, val, chan_out)
int ad, val;
CHAN *chan_out;
```

This function writes the value **val** into the word of memory at address **ad** in a neighbouring transputer. In order to be able to do this, the neighbour transputer must have been recently reset but *not* bootstrapped. In this special state, the transputer processor executes special firmware implementing a 'peek and poke' protocol described in the INMOS data sheets[10,11] and the *Compiler Writer's Guide*[12]. The function returns a non-zero value if the 'peek' operation succeeds.

The neighbouring transputer is connected to the one on which the **boot_poke** function is executed by an INMOS link, with which is associated an output channel **chan_out**. If that link does not lead to another transputer, or if the other transputer is not executing the 'peek and poke' firmware, the **boot_poke** function will time out after 30 ticks of the transputer timer associated with the current thread's priority. This timeout period is around 2mS for a non-urgent thread. If **boot_poke** times out, it returns zero.

calloc allocates and clears an area of memory

```
char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

calloc returns a pointer to enough space for **nelem** objects of size **elsize**, or **NULL** if the request cannot be satisfied. The storage is initialised to zero.

ceil ceiling function

```
#include <math.h>
double ceil(x)
double x;
```

ceil returns the smallest integer not less than **x**.

cfree deallocates the space allocated by **calloc** or **realloc**

```
cfree(ptr)
char *ptr;
```

cfree frees the space pointed to by **ptr**, where **ptr** was originally obtained by a call to **calloc** or **realloc**.

chan_in_byte input a byte from a channel

```
#include <chan.h>
chan_in_byte(b, chan)
char *b;
CHAN *chan;
```

This function reads a single-byte message from the channel pointed to by **chan** into the character variable pointed to by **b**.

chan_in_byte_t input a byte from a channel, or timeout

```
#include <chan.h>
chan_in_byte_t(b, chan, timeout)
char *b;
CHAN *chan;
int timeout;
```

This function attempts to read a single-byte message from the channel pointed to by **chan** into the character variable pointed to by **b**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_init initialise a channel word

```
#include <chan.h>
chan_init(chan)
CHAN *chan;
```

This function initialises the channel word pointed to by its **chan** argument. This operation consists of writing the special value **NotProcess_P** (**MOSTNEG INT**) into the channel word; this indicates that no threads are currently attempting to communicate through this channel.

All channel words (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 transputer processor to crash.

Note that the channel words 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 input a message from a channel

```
#include <chan.h>
chan_in_message(l, b, chan)
int l;
char *b;
CHAN *chan;
```

This function reads a message of length **l** bytes from the channel pointed to by **chan** into the variable pointed to by **b**.

chan_in_message_t input a message from a channel, or timeout

```
#include <chan.h>
chan_in_message_t(l, b, chan, timeout)
int l, timeout;
char *b;
CHAN *chan;
```

This function attempts to read a message of length **l** bytes from the channel pointed to by **chan** into the variable pointed to by **b**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_in_word input a word from a channel

```
#include <chan.h>
chan_in_word(w, chan)
int *w;
CHAN *chan;
```

This function reads a four-byte message from the channel pointed to by **chan** into the integer variable pointed to by **w**.

chan_in_word_t input a word from a channel, or timeout

```
#include <chan.h>
chan_in_word_t(w, chan, timeout)
int *w, timeout;
CHAN *chan;
```

This function attempts to read a four-byte message from the channel pointed to by **chan** into the integer variable pointed to by **w**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_out_byte output a byte to a channel

```
#include <chan.h>
chan_out_byte(b, chan)
char b;
CHAN *chan;
```

This function sends a single-byte message consisting of the value **b** to the channel pointed to by **chan**.

chan_out_byte_t output a byte to a channel, or timeout

```
#include <chan.h>
chan_out_byte_t(b, chan, timeout)
char b;
CHAN *chan;
int timeout;
```

This function attempts to send a single-byte message consisting of the value **b** to the channel pointed to by **chan**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_out_message output a message to a channel

```
#include <chan.h>
chan_out_message(l, b, chan)
int l;
char *b;
CHAN *chan;
```

This function sends a message of length **l** bytes from the variable pointed to by **b** to the channel pointed to by **chan**.

chan_out_message_t output a message to a channel, or timeout

```
#include <chan.h>
chan_out_message_t(l, b, chan, timeout)
int l, timeout;
char *b;
CHAN *chan;
```

This function attempts to send a message of length **l** bytes from the variable pointed to by **b** to the channel pointed to by **chan**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_out_word output a word to a channel

```
#include <chan.h>
chan_out_word(w, chan)
int w;
CHAN *chan;
```

This function sends a four-byte message consisting of the value **w** to the channel pointed to by **chan**.

chan_out_word_t output a word to a channel, or timeout

```
#include <chan.h>
chan_out_word_t(w, chan, timeout)
int w, timeout;
CHAN *chan;
```

This function attempts to send a four-byte message consisting of the value **w** to the channel pointed to by **chan**. If the communication does not take place within **timeout** ticks of the timer associated with the priority of the current thread, the function will terminate and return zero. If the communication succeeds within the timeout interval, the function will return a non-zero value.

chan_reset reset a channel

```
#include <chan.h>
char *chan_reset(chan)
CHAN *chan;
```

This function resets the channel pointed to by **chan**. If the channel is associated with an INMOS link, then the hardware of that link is reset as well.

If a thread was attempting to communicate on the channel at the time of the reset, then a handle to that thread (which is now suspended) will be returned as the result of **chan_reset**. This handle can be used to restart the suspended thread at a later date by passing it to the function **thread_restart**.

If the channel was idle at the time of the reset (i.e. if no thread was attempting to communicate on it) then the value **NotProcess_P** (**MOSTNEG INT**) will be returned.

clearerr† clear stream errors

```
#include <stdio.h>
clearerr(stream)
FILE *stream;
```

clearerr resets any error indication on the named stream. It is implemented as a macro and therefore may not be redeclared.

clock return processor time used

```
#include <time.h>
clock_t clock()
```

The **clock** function determines the processor time used. It returns the elapsed time in seconds 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** and **CLK_TCK** is 1.

The time in seconds is the value returned divided by the value of the macro **CLK_TCK** (also defined by **<time.h>**).

close close a file

```
int close(fildes)
int fildes;
```

Given a file descriptor (**fildes**) as returned by **open** or **creat**, **close** closes the associated file, i.e. breaks the connection between the file descriptor (a small integer) and the file itself. A close of all files is automatic on exit, but since there is a limit on the number of files which may be open at once, **close** is necessary for programs which deal with many files.

Zero is returned if a file is closed, −1 is returned for an unknown file descriptor.

cos cosine function

```
#include <math.h>
double cos(x)
double x;
```

cos returns the cosine of its radian argument.

cosh hyperbolic cosine function

```
#include <math.h>
double cosh(x)
double x;
```

cosh returns the hyperbolic cosine of its argument.

creat create a new file

```
creat(name, mode)
char *name;
int mode;
```

creat creates a new file or prepares to rewrite an existing file called **name**, given as the address of a NUL-terminated string. The **mode** argument is currently ignored, but should be given by the caller for portability.

exit terminate execution

```
exit(status)
int status;
```

exit is the normal means of terminating program execution. **exit** closes all the process's files

This call never returns.

exp e^x function

```
#include <math.h>
double exp(x)
double x;
```

exp returns the exponential function of **x**.

exp returns a huge value when the correct value would overflow; **errno** is set to '**ERANGE**'.

fabs floating absolute value

```
#include <math.h>
double fabs(arg)
double arg;
```

fabs returns the absolute value of **arg**.

fclose close a file

```
#include <stdio.h>
int fclose(stream)
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 **EOF** if **stream** is not associated with an output file, or if buffered data cannot be transferred to that file.

fdopen open a stream

```
#include <stdio.h>
FILE *fdopen(fildes, type)
char *type;
int fildes;
```

fdopen associates a stream with a file descriptor obtained from **open** or **creat**.

type is a character string specifying the way in which the file is to be opened. Refer to the description of **fopen** (page 179) for a full description of the **type** string.

The **type** of the stream must agree with the way the file was opened.

feof† is stream at end of file?

```
#include <stdio.h>
int feof(stream)
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, and therefore cannot be redeclared.

ferror† tests for stream errors

```
#include <stdio.h>
int ferror(stream)
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 flush stream buffer

```
#include <stdio.h>
int fflush(stream)
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 read a character from a stream

```
#include <stdio.h>
int fgetc(stream)
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.

fgets read a string from a stream

```
#include <stdio.h>
char *fgets(str, n, stream)
char *str;
int n;
FILE *stream;
```

fgets reads **n** – 1 characters, or up to a newline character, whichever comes first, from the **stream** into the string **str**. The last character read into **str** is followed by a NUL character. **fgets** returns its first argument.

fgets returns **NULL** on end of file or error

Note that **fgets** behaves differently from **gets** (q.v.) with respect to any terminating newline character: **fgets** keeps the newline, **gets** deletes it from the string.

fileno† stream status enquiry

```
#include <stdio.h>
int fileno(stream)
FILE *stream;
```

fileno returns the low-level I/O 'file descriptor' associated with the stream, see **open**. It is implemented as a macro. (Standard I/O is implemented by calls on the low-level functions).

floor floor function

```
#include <math.h>
int floor(x)
double x;
```

floor returns the largest integer not greater than **x**.

fmod calculate the floating-point remainder of the argument division

```
#include <math.h>
double fmod(x, y)
double x, y;
```

fmod returns the remainder from **x/y**

fopen opens a file

```
#include <stdio.h>
FILE *fopen(filename, type)
char *filename, *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. **fopen** returns the pointer **NULL** if **filename** cannot be accessed in the way requested.

type is a character string made up of the following parts:

- 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.
- 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**.
- An optional specification of whether the file is to be a normal text file ('**t**') or a binary file ('**b**'). If this specifier is omitted, the value of the **_fmode** variable is used to determine the mode of access. The default is **O_TEXT**; files are opened as text files.

The second and third parts of the **type** string may appear in any order. For example, **"r+b"** and **"rb+"** are equivalent. Some examples of possible values for **type** are now given, along with a description of their interpretation.

"r"	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.

fprintf formatted output

```
#include <stdio.h>
int fprintf(stream, format[, arg1 [, arg2 [...]]])
FILE *stream;
char *format;
```

fprintf writes its output on the specified **stream** (by calling **putc**). **fprintf** converts, formats and outputs the arguments **argi** under control of its **format** argument. The **format** argument is a character string which 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 **arg** value.

Each conversion specification is introduced by the character '%'. Following the '%' there may be (in the given order):

- an optional minus sign '–', which specifies *left justification* of the converted value in the indicated field;
- an optional digit string specifying a *field width*; if the converted value has fewer characters then the field width it will be blank padded on the left (or right if left justification indicator – has been given) to make up the field width; if the field width begins with a zero, zero padding will be performed instead of blank padding.
- an optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point for '**e**' and '**f**' format conversion, or the maximum number of characters to be output from a string;
- the character '**l**' (lowercase 'L'), specifying that a following '**d**', '**o**' '**x**' or '**u**' corresponds to a long integer **arg**. (A capitalised conversion code, such as 'D', has the same effect).
- A character which indicates the type of conversion to be applied.

A field width or precision may be specified as '*' instead of a digit string, in which case a corresponding integer **arg** is used as the field width or precision respectively.

The conversion characters and their meanings are:

'**d**' The integer **arg** is converted to decimal notation.

'**o**' The integer **arg** is converted to octal notation.

'**x**' The integer **arg** is converted to hexadecimal notation.

'**f**' The float or double **arg** is converted at decimal notation in the form '[–]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

'**e**' The float or double **arg** is converted into the form '[–]d.ddde[±]dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is not specified, 6 digits are produced.

'g' The float or double **arg** is output in style 'd', 'f' or 'e', whichever gives full precision in minimum space.

'c' The (**char**) **arg** is printed. NUL characters are ignored.

's' **arg** 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 a NUL are printed.

'u' The unsigned integer **arg** is converted to decimal and output.

'%' Print a '%'; no argument is converted

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by **fprintf** are printed by **putc** (q.v.)

Note that fields wider than 128 characters do not work.

fprintf returns the number of characters output, or a negative value if an output error occurred.

fputc write a character to a stream

```
#include <stdio.h>
int fputc(cval, stream)
FILE *stream;
int cval;
```

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 write a string to a stream

```
#include <stdio.h>
fputs(str, stream)
char *str;
FILE *stream;
```

fputs copies the NUL-terminated string **str** to the specified output **stream**. The NUL character which terminates the string is *not* written to the stream.

Note that **fputs** is inconsistent with **puts**, which appends a newline to the output string.

fread buffered binary input

```
#include <stdio.h>
int fread(ptr, size, nitems, stream)
FILE *stream;
int nitems, size;
char *ptr;
```

fread reads into a block beginning at **ptr**, **nitems** of data of the type of ***ptr** from the specified input **stream**. **size** will be the value of **sizeof(*ptr)**. It returns the number of items actually read.

fread returns zero on end of file or error.

free deallocates the space allocated by **malloc** or **realloc**

```
free(ap)
char *ap;
```

free frees the space pointed to by **ap**, where **ap** was originally obtained by a call to **malloc** or **realloc**.

freopen open a stream

```
#include <stdio.h>
FILE *freopen(filename, type, stream)
char *filename, *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 preopened 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** (page 179) for a full description of the **type** string.

freopen returns the pointer **NULL** if **filename** cannot be accessed.

frexp split floating-point number into separate parts

```
#include <math.h>
double frexp(value, exp)
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 pointed to by ***exp**.

fscanf formatted input

```
#include <stdio.h>
int fscanf(stream, format[, ptr1 [, ptr2 [...]]])
FILE *stream;
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 its arguments.

The format string usually contains conversion specifications which are used to direct interpretation of input sequences. The control string may contain:

- Blanks, tabs or newlines, which match optional white space in the input.
- An ordinary character (not %) which must match the next character of the input stream.
- Conversion specifications: the character '%' followed by an optional assignment suppressing character, '*', followed by an optional numerical maximum field width and finally a conversion character.

  A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding **ptr** argument, unless assignment suppression was indicated by '*'. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal;

'%' a single '%' is expected in the input at this point; no assignment is done.

'd' a decimal integer is expected; the corresponding argument should be an integer pointer.

'o' an octal integer is expected; the corresponding argument should be an integer pointer.

'x' a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

's' a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

'c' a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, use '%1s'. If a field width is given, the corresponding argument should point to a character array, and the indicated number of characters is read.

'e' a floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is 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.

'[' Indicates a string not to be delimited by space characters. The left bracket is followed in the format string by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex ('^'), the input field is all the characters until the first character not in the set between the brackets; if the first character after the left bracket is '^', the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters 'd', 'o' and 'x' may be capitalised or preceded by 'l' (lowercase 'L') to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters 'e' or 'f' may be capitalised or preceded by a 'l' (lowercase 'L') to indicate a pointer to double rather than to float. The conversion characters 'd', 'o' and 'x' may be preceded by 'h' to indicate a pointer to short rather than to int.

fscanf returns the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned on end of input; note that this is different from zero, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

EOF is returned on end of input.

fseek reposition a stream

```
#include <stdio.h>
int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, the current position or the end of the file, depending on whether ptrname has the value 0, 1 or 2.

fseek undoes any effects of ungetc.

fseek returns −1 for improper seeks.

ftell stream position enquiry

```
#include <stdio.h>
long ftell(stream)
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 bytes.

fwrite buffered binary output

```
#include <stdio.h>
int fwrite(ptr, size, nitems, stream)
FILE *stream;
int nitems, size;
char *ptr;
```

fwrite appends at most **nitems** of data of the type of ***ptr** beginning at **ptr** to the specified output **stream**. **size** will be the value of **sizeof(*ptr)**. It returns the number of items actually written.

Zero is returned on end of file or error conditions.

getc† read a character from a stream

```
#include <stdio.h>
int getc(stream)
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† read a character from standard input

```
#include <stdio.h>
int getchar()
```

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.

gets read string from standard input

```
#include <stdio.h>
char *gets(str)
char *str;
```

gets reads a string into **str** from the standard input stream **stdin**. The 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** is inconsistent with **fgets** (q.v.) in its treatment of the terminating newline character: **gets** deletes the newline, **fgets** keeps it.

index find character in string

```
char *index(s, c)
char *s, c;
```

This function searches the string **s** for the first occurrence of character **c**, and returns a pointer to it. If **c** does not occur in **s**, a null pointer is returned.

isalnum† is character alphanumeric?

```
#include <ctype.h>
int isalnum(cval)
int cval;
```

cval is a letter or a digit, 0 otherwise.

isalpha† is character alphabetic?

```
#include <ctype.h>
int isalpha(cval)
int cval;
```

cval is a letter, 0 otherwise.

isascii† is argument an ASCII character?

```
#include <ctype.h>
int isascii(cval)
int cval;
```

cval is an ASCII character (code less than 200 octal).

isatty is file descriptor a terminal?

```
#include <stdio.h>
int isatty(fildes)
int fildes;
```

isatty returns 1 if the file descriptor **fildes** is associated with a terminal device, 0 otherwise.

`iscntrl`† ASCII control character?

```
#include <ctype.h>
int iscntrl(cval)
int cval;
```

`cval` is an ASCII control character, 0 otherwise.

`isdigit`† is argument a digit?

```
#include <ctype.h>
int isdigit(cval)
int cval;
```

`cval` is a digit, 0 otherwise.

`isgraph`† printing ASCII character other than space?

```
#include <ctype.h>
int isgraph(cval)
int cval;
```

`cval` is a printing character, codes 41 octal ('!') through 176 octal ('~'). Returns 0 otherwise.

`islower`† is character lowercase?

```
#include <ctype.h>
int islower(cval)
int cval;
```

`cval` is a lowercase letter, 0 otherwise.

`isprint`† printing ASCII character?

```
#include <ctype.h>
int isprint(cval)
int cval;
```

`cval` is a printing character, codes 40 octal (space) through 176 octal ('~'). Returns 0 otherwise.

ispunct† punctuation character?

```
#include <ctype.h>
int ispunct(cval)
int cval;
```

cval is a punctuation character (neither control nor alphanumeric), 0 otherwise.

isspace† white space character?

```
#include <ctype.h>
int isspace(cval)
int cval;
```

cval is a space, horizontal or vertical tab, carriage return, newline or form feed character, 0 otherwise.

isupper† is character uppercase?

```
#include <ctype.h>
int isupper(cval)
int cval;
```

cval is an uppercase letter, 0 otherwise.

isxdigit† printing hexadecimal digit?

```
#include <ctype.h>
int isxdigit(cval)
int cval;
```

cval is a printing hexadecimal digit, 0 otherwise.

`ldexp` calculate x^{exp}

```
#include <math.h>
double ldexp(x, exp)
double x;
int exp;
```

`ldexp` returns the result of `x` multiplied by the value of two raised to the power `exp`.

`log` calculates $\log_e x$

```
#include <math.h>
double log(x)
double x;
```

`log` returns the natural logarithm of `x`.

`log` returns 0 when `x` is zero or negative; the `extern int` variable `errno` is set to `EDOM`. `EDOM` is defined in the header file `<errno.h>`

`log10` calculates $\log_{10} x$

```
#include <math.h>
double log10(x)
double x;
```

`log10` returns the base-ten logarithm of `x`.

`log10` returns 0 when `x` is zero or negative; the `extern int` variable `errno` is set to `EDOM`. `EDOM` is defined in the header file `<errno.h>`.

longjmp non-local goto

```
#include <setjmp.h>
longjmp(env, val)
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 which 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.

lseek move read/write pointer

```
long lseek(fildes, offset, whence)
long offset;
int fildes, whence;
```

The file descriptor refers to a file open for reading and writing. The read (resp. write) pointer for the file is set as follows:

whence = 0 : the pointer is set to **offset** bytes.

whence = 1 : the pointer is set to its current location plus **offset**.

whence = 2 : the pointer is set to the size of the file plus **offset**.

The returned value is the resulting pointer location.

−1 is returned for an undefined file descriptor or a seek to a position before the beginning of the file.

lseek is a no-op on devices (e.g. the VDU or keyboard) which are not disk files.

malloc allocates the specified number of contiguous bytes of memory

```
char *malloc(nbytes)
unsigned nbytes;
```

malloc allocates space for an object whose size is specified by **nbytes**. The function returns a pointer to the start of the allocated space. If the space cannot be allocated, the **malloc** function returns a **NULL** pointer.

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 the **calloc** function has two arguments compared to **malloc**'s one. Thus, calls to **malloc** must be rewritten from **malloc(n)** to **calloc(n,1)**.

memcpy memory block move

```
#include <string.h>
void *memcpy(s1, s2, n)
void *s1, *s2;
unsigned int n;
```

memcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. If copying takes places between objects that overlap, the behaviour is undefined.

memcpy returns the value of **s1**.

memset fill object with repeated byte value

```
#include <string.h>
void *memset(ptr, cval, num)
void *ptr;
int cval;
unsigned int num;
```

The **memset** function copies the value of **cval** (converted to an **unsigned char**) into each of the first **num** characters of the object pointed to by **ptr**.

The **memset** function returns the value of **ptr**.

modf split argument into integral and fractional parts

```
#include <math.h>
double modf(value, iptr)
double value, *iptr;
```

modf split **value** into its integral and fractional parts. The function returns the signed fractional part and the integral part is pointed to by ***iptr**

net_receive receive a flood-filled network message

```
#include <net.h>
int net_receive(packet, complete)
char *packet;
int *complete;
```

This function can be called by tasks participating in a flood-filled application to receive a message from the network.

The next (or only) packet of the message being received is read into the buffer pointed to by **packet**.

If **net_receive** is called by the master task it reads the next available result packet returned by a worker task; if it is called from a worker task, it reads the next work packet sent out by the master.

The size of the packet (in bytes) is returned as the result of the function.

If the **packet** is the final or only packet of the message, the location pointed to by **complete** will be set to 1; otherwise it is set to 0 and the receiving task must repeatedly call **net_receive** to read the remaining part of the message.

No more than **NET_MAX_PACKET_LENGTH** bytes will be read into the **packet** buffer. Less space may be allocated if it is certain that the sending task will not send messages longer than some smaller limit (for example, if only fixed-length messages are being used).

net_send send a flood-filled network message

```
#include <net.h>
int net_send(nbytes, packet, complete)
char *packet;
int nbytes, complete;
```

This function can be called by tasks participating in a flood-filled application to send a message into the network.

If **net_send** is called by the master task, the message packet is sent to any free worker task; if the function is called by a worker task, the packet is sent back to the master task.

nbytes is the number of bytes of data in the buffer pointed to by **packet**.

If **nbytes** is less than zero or greater than **NET_MAX_PACKET_LENGTH** (defined in version 2.0 of Parallel C by **<net.h>** to be 1024 bytes) no message is sent and the function returns a negative value.

Otherwise the function returns the number of bytes sent, which will be **nbytes** if no error occurs.

If a message longer than **NET_MAX_PACKET_LENGTH** has to be sent, it must be broken up into a number of packets, each smaller than this limit.

If **complete** is 0, the argument **packet** is regarded as part of a larger message; a circuit to the destination task is held open until the last packet of the message has been sent. The final (or only) packet of a message is marked by setting **complete** equal to 1.

The routing software guarantees that multiple packets sent in this way are always received by the destination task in the same order they were sent.

In normal use, packets will be smaller than 1024 bytes and **complete** will always be given the value 1. Sending very long packets can clog up the network, blocking packets being delivered to other nodes.

open open for reading or writing

```
int open(name, mode)
char *name;
int mode;
```

open opens the file **name** for reading (**mode** = 0), writing (**mode** = 1) or for both reading and writing (**mode** = 2). **name** is the address of a string of ASCII characters representing a file name, terminated by an ASCII NUL character. The file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

The value −1 is returned if the file does not exist or is unreadable or if too many files are already open.

par_free deallocate space allocated by **par_malloc**

```
#include <par.h>
par_free(ap)
char *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**.

par_fprintf formatted output

```
#include <stdio.h>
#include <par.h>
int par_fprintf(stream, format[, arg1[, arg2[...]]])
FILE *stream;
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**.

par_printf formatted output on **stdout**

```
#include <par.h>
int par_printf(format [ , arg1 [ , arg2 [...]]])
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**.

par_malloc allocate the specified number of contiguous bytes of memory

```
#include <par.h>
char *par_malloc(nbytes)
unsigned nbytes;
```

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**.

pow calculates x^y

```
#include <math.h>
double pow(x, y)
double x, y;
```

pow returns the value of **x** raised to the power of **y**.

printf formatted output on **stdout**

```
#include <stdio.h>
int printf(format [, arg1 [, arg2 [...]]])
char *format;
```

printf writes output to the standard output stream, **stdout**. It returns the number of characters which 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**.

```
printf(...);
```

is equivalent to

```
fprintf(stdout, ...);
```

putc† writes a single character to a file

```
#include <stdio.h>
int putc(cval, stream)
char 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, **putc(c, *f++);** does not work sensibly.

putchar† write a character to standard output

```
#include <stdio.h>
int putchar(cval)
int cval;
```

putchar(cval) is a macro defined as **putc(cval,stdout)**. I.e. the character **cval** is written to the standard output stream, **stdout** (normally the VDU).

EOF is returned on error.

puts write string to standard output

```
#include <stdio.h>
puts(pstr)
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 VDU.

puts appends a newline to the output string but **fputs** (q.v.) does not.

putw write an integer to standard output

```
#include <stdio.h>
int putw(ival, stream)
FILE *stream;
int ival;
```

putw outputs an integer value to the standard output stream in a format which can be read in again by the standard input function **getw**.

putw returns the integer value written.

putw neither assumes nor causes special alignment in the file.

EOF is returned if a write error occurs.

rand pseudo-random number generator

```
int rand()
```

rand function returns successive pseudo-random integers in the range 0 to 32767.

read read from file

```
int read(fildes, buffer, nbytes)
char *buffer;
int fildes, nbytes;
```

A file descriptor is an integer returned by a successful call on **open** or **creat**. **buffer** is the location of **nbytes** contiguous bytes into which the input will be placed. It is not guaranteed that all **nbytes** bytes will be read; for example if the file descriptor refers to the keyboard at most one line will be returned. In any event, the number of characters actually read is returned.

Zero is returned when the end of the file has been reached. If the read was unsuccessful for any other reason, −1 is returned. Many conditions may cause errors: physical I/O errors, bad buffer address etc.

realloc changes the size of an area allocated by **malloc** or **realloc**

```
char *realloc(ptr, size)
char *ptr;
unsigned size;
```

realloc changes the size of the object pointed to by **ptr** to the size specified 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.

remove removes a file from the file system

```
#include <stdio.h>
int remove(filename)
char *filename;
```

remove function causes the file whose name is the string pointed to by filename to be removed. Subsequent attempts to open the file will fail, unless it is created anew.

remove returns −1 if the file cannot be removed.

rewind reposition stream to beginning

```
#include <stdio.h>
int rewind(stream)
FILE *stream;
```

rewind(stream) is equivalent to **fseek(stream, 0L, 0)**. It repositions **stream** to the first byte of the associated file (byte 0). It is a no-op if the stream is associated with a device rather than a file (e.g. the keyboard or the VDU).

rewind returns −1 on failure.

rindex find character in string

```
char *rindex(s, c)
char *s, c;
```

This function searches the string **s** for the last occurrence of character **c**, and returns a pointer to it. If **c** does not occur in **s**, a null pointer is returned.

scanf formatted input from **stdin**

```
#include <stdio.h>
int scanf(format [, ptr1 [, ptr2 [ ... ]]])
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 **ptr** arguments.

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, ...);
```

scanf returns **EOF** on end of input, and a short count for missing or illegal data items.

sema_init initialise a semaphore

```
#include <sema.h>
sema_init(s, v)
int v;
SEMA *s;
```

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 **static** or external semaphore is left uninitialised, it defaults to an empty queue of threads and an initial value of 0. If an **auto** semaphore is left uninitialised, the first **sema_signal** or **sema_wait** operation on the semaphore will cause the transputer system to behave unpredictably.

sema_signal perform a *signal* operation on a semaphore

```
#include <sema.h>
sema_signal(s)
SEMA *s;
```

If there are threads waiting for the semaphore pointed to by **s**, one of them 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.

If there are no threads waiting for the semaphore pointed to by **s**, its value will be increased by 1.

Note that any particular semaphore must be accessed only by threads executing at one particular priority. For example, it would be acceptable for a set of 'urgent' threads to synchronise through a semaphore, or for a set of 'not urgent' threads to do this, but not for a mixture of threads executing at different priorities. Threads executing at different priorities can synchronise by passing messages along channels.

sema_signal_n perform *n* *signal* operations on a semaphore

```
#include <sema.h>
sema_signal_n(s, n)
int n;
SEMA *s;
```

This function calls the function **sema_signal** *n* times, in sequence.

sema_wait perform a *wait* operation on a semaphore

```
#include <sema.h>
sema_wait(s)
SEMA *s;
```

If the value of the semaphore pointed to by **s** is not zero, its value is decreased by 1.

If the value of the semaphore is 0, the value is left unchanged and the current thread is added to the list of threads waiting for the semaphore, and paused. It will be resumed by some future call on **sema_signal**.

Programs should not rely on any relationship between the order in which threads start to wait on a semaphore and the order in which they will be resumed. At present, threads are simply 'pushed down' onto the list of waiting processes, so that the last thread to start waiting on a semaphore will be the first to be resumed.

Note that any particular semaphore must be accessed only by threads executing at one particular priority. For example, it would be acceptable for a set of 'urgent' threads to synchronise through a semaphore, or for a set of 'not urgent' threads to do this, but not for a mixture of threads executing at different priorities. Threads executing at different priorities can synchronise by passing messages along channels.

sema_wait_n perform *n* *wait* operations on a semaphore

```
#include <sema.h>
sema_wait_n(s, n)
int n;
SEMA *s;
```

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.

serv_filter start run-time library protocol filter threads

```
#include <serv.h>
serv_filter (norm_in,norm_out, wide_in,wide_out)
CHAN *norm_in, *norm_out, *wide_in, *wide_out;
```

A historical problem involving first-silicon T414A transputers was solved by making the protocol used by the full run-time library different to the documented protocol when a communication is performed across the transputer links.

When the standard harness is used (**mainent.c4x** or **mainent.c8x**) there is no 'problem' as the protocol used by the **iserver** is different to that used by the full run-time library, which does not have this problem. The standard harness has a protocol converter (written in occam) which performs the conversion between the protocol used by the full run-time library and the protocol used by the **iserver**.

In configuring programs the mismatch of protocol between the full run-time library and the **iserver** is handled by the purpose-built **filter** task (again by a protocol converter written in occam).

This function allows a program to start a pair of threads which perform this protocol conversion. The workspace for these threads is roughly 1200 bytes in total; this is allocated from the heap. After the filter threads have been started, control is returned to the caller.

norm_in and **norm_out** are connected to the 'normal' task (i.e. the one using the standard unmodified protocol used by the run-time library) while **wide_in** and **wide_out** are connected to the task using the T414A-tolerant variant protocol. The latter will normally be the pair of physical links connected to the host.

The sense of the in/out labels on the arguments to this function is from the point of view of the tasks to which the filter is being attached. For example, **norm_in** is an input channel to the normal-protocol task; it will therefore be an output channel to the task containing the **serv_filter** call.

Note that the maximum size of a variable-length data item which may be passed through the filter in either direction is 512 bytes.

setbuf assign buffering to a stream

```
#include <stdio.h>
setbuf(stream, buf)
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 streams directed to the VDU and the standard error stream **stderr** are normally not buffered.

setjmp save environment for longjmp

```
#include <setjmp.h>
int setjmp(env)
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 which 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.

sin sine function

```
#include <math.h>
double sin(x)
double x;
```

sin returns the sine of its radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

sinh hyperbolic sine function

```
#include <math.h>
double sinh(x)
double x;
```

sinh returns the hyperbolic sine of its argument.

sprintf formatted output to a string

```
#include <stdio.h>
int sprintf(pstr, format [, arg1[, arg2[...]]])
char *pstr, *format;
```

sprintf writes formatted output into a character array via a pointer **pstr** supplied by the caller. It returns the number of characters written into the array.

The meaning of **format** and the **arg** pointers is as for **fprintf**.

The output string **pstr** is automatically terminated by a NUL character. Note that this terminator is *not* included in the character count returned by **sprintf**.

sqrt calculates $\sqrt{x}$

```
#include <math.h>
double sqrt(x)
double x;
```

sqrt returns the square root of **x**.

sqrt returns zero when **x** is negative; **errno** is set to **EDOM**.

srand new seed for rand function

```
srand(seed)
unsigned int seed;
```

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 formatted input from string

```
#include <stdio.h>
int sscanf(pstr, format [, ptr1[, ptr2[...]]])
char *pstr, *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 **ptr** arguments.

The exact meaning of the arguments to **sscanf** is the same as for **fscanf**.

strcat concatenates two strings

```
#include <string.h>
char *strcat(s1, s2)
char *s1, *s2;
```

strcat appends a copy of string **s2** to the end of string **s1**. A pointer to the NUL-terminated result is returned.

strchr find a specified character in a string

```
#include <string.h>
char *strchr(pstr, cval)
char *pstr;
int cval;
```

strchr locates the first occurrence of cval (converted to a char) in the string pointed to by pstr. The terminating null 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 string compare

```
#include <string.h>
int strcmp(s1, s2)
char *s1, *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**.

strcpy string copy

```
#include <string.h>
char *strcpy(s1, s2)
char *s1, *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 find length of string that does not contain specified characters

```
#include <string.h>
unsigned strcspn(s1, s2)
char *s1, *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 null character is not considered part of the s2. The function returns the length of the part.

strlen string length

```
#include <string.h>
int strlen(pstr)
char *pstr;
```

strlen returns the number of non-NUL characters in **pstr**.

strncat string concatenate

```
#include <string.h>
char *strncat(s1, s2, num)
char *s1, *s2;
int num;
```

strncat appends a copy of string **s2** to the end of string **s1**. It copies at most **num** characters. A pointer to the NUL-terminated result is returned.

strncmp string compare

```
#include <string.h>
int strncmp(s1, s2, num)
char *s1, *s2;
int 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 are looked at.

strncpy string copy

```
#include <string.h>
char *strncpy(s1, s2, num)
char *s1, *s2;
int num;
```

strncpy copies string **s2** to **s1**. Exactly **num** characters are copied: **s2** is truncated or NUL-padded as required. The target may not be NUL terminated if the length of **s2** is **n** or more. **s1** is returned.

strspn find length of string which contains specified characters

```
#include <string.h>
unsigned strspn(s1, s2)
char *s1, *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.

strtok break strings into tokens

```
#include <string.h>
char *strtok(s1, s2)
char *s1, *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 convert string to **long int**

```
#include <stdlib.h>
long int strtol (nptr, endptr, base)
char *nptr, **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 null 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–z** are ascribed the values 10–35. Only those characters which are representations of values less than **base** are allowed. If **base** has the value 16, the characters **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**), 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 trap 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 0 is returned and **errno** is set to **EDOM**. In this case, the object pointed to by **endptr** is set to the value of **nptr** (unless **endptr** is 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 convert string to **unsigned long int**

```
#include <stdlib.h>
unsigned long int strtoul (nptr, endptr, base)
char *nptr, **endptr;
int base;
```

This function operates in the same way as **strtol**, except:

- It returns an **unsigned long int**;
- + and − are not accepted as part of the subject string;
- In the event of an overflow being trapped, the value returned is always **ULONG_MAX**.

tan tangent function

```
#include <math.h>
double tan(x)
double x;
```

tan returns the tangent of its radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

tanh hyperbolic tangent function

```
#include <math.h>
double tanh(x)
double x;
```

tanh returns the hyperbolic tangent of its argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

thread_create create a simple thread

```
#include <thread.h>
char *thread_create(fn, wssize, nargs,
                                arg1, ..., argn)
void (*fn)();
int wssize;
int nargs;
int arg1, ..., argn;
```

The function **fn** is started as a new thread, running at the same priority as the current thread, with a workspace of **wssize** bytes. This workspace is taken from the heap and a pointer to it is returned from **thread_create** so that, if desired, the workspace can be returned to the heap (using **par_free**) once the thread is known to have stopped. If there is insufficient heap space remaining to create the requested workspace, this function will return **NULL**.

The **nargs** arguments **arg1**...**argn** will be passed on to the thread's function **fn** as its arguments; any number of arguments may be supplied here as long as the correct number is recorded in **nargs**.

This function is a shorthand way of calling the more general thread creation function **thread_start** in the most usual circumstances.

If **thread_create** is used to create a new thread when a task is executing with combined stack and data areas (no **S** option is specified when **iboot** is used to make a single transputer program executable, or when configuring a task with the **data=** attribute option), care must be taken when allocating an area of storage from the heap.

In these circumstances, the heap allocation functions check that the amount of space you request is actually available by comparing the top of the heap with the current workspace pointer. For threads started with **thread_create**, this will be within the heap and the check will therefore fail; the allocation function will return a null pointer to indicate that no more space is available in the heap area.

If the separate stack and data storage allocation model is used, the heap allocation functions check allocations against the true end of the heap area, and this problem therefore cannot arise.

There are several ways to avoid this problem:

- Arrange your code so that threads started using **thread_create** neither allocate space from the heap or (equivalently) themselves call **thread_create**.
- Configure the task in which such operations are performed in the separate stack and heap mode, using the STACK and HEAP attributes of the configuration language TASK statement.

thread_deschedule make current thread momentarily unable to execute

```
#include <thread.h>
thread_deschedule()
```

This function causes a thread to become momentarily unable to execute (usually for one timer tick); this will cause it to be *descheduled* from the processor, thus allowing some other thread to resume execution in its place. Eventually, the thread which called **thread_deschedule** will resume.

This function can be used by a thread performing some background computation to prevent it from 'hogging' the processor to the detriment of other threads executing at the same priority level. In effect, a priority level even less urgent than **THREAD_NOTURG** can be achieved for use by threads performing long-term CPU-intensive tasks whose results are not expected to be immediately required.

thread_priority return current thread's priority

```
#include <thread.h>
int thread_priority()
```

This function returns the priority of the current thread, which will be either **THREAD_URGENT** or **THREAD_NOTURG**.

thread_restart restart a thread given its workspace

```
#include <thread.h>
thread_restart(p)
char *p;
```

p should be a pointer to the workspace of a thread which is known to be stopped. The effect of this function is to restart that thread from where it left off.

This function can be used to restart threads which have been stopped because the channel on which they were attempting to communicate has been reset using a call to **chan_reset**, which returns a handle suitable for use by **thread_restart**.

thread_start start a general thread

```
#include <thread.h>
thread_start(fn, ws, wssize, flags,
                    nargs, arg1, ..., argn)
void (*fn)();
char *ws;
int wssize;
int nargs;
int arg1, ..., argn;
```

This function starts a new thread based on the function **fn**. The new thread uses the area **ws** as its workspace. The size of the workspace (**wssize**) is a number of bytes.

The new thread will stop either when it executes the function **thread_stop**, or when **fn** returns.

The **flags** argument is a set of attributes for the new thread. At present, the only attribute available is the thread's priority, which should be either **THREAD_URGENT** or **THREAD_NOTURG**. Normally, new threads should be started at the same priority as the current thread. This is achieved by passing the result of the function **thread_priority** described below as the value of this argument. Other than the priority specification, all bits in the **flags** argument are reserved, and should be 0.

The **nargs** arguments **arg1**...**argn** will be passed on to the thread's function **fn** as its arguments; any number of arguments may be supplied here as long as the correct number is recorded in **nargs**.

See also the description of **thread_create**, which simplifies thread creation by starting a thread at the current priority and allocates the thread's workspace from the heap.

thread_stop stop the current thread

```
#include <thread.h>
thread_stop()
```

This function stops the current thread. The current thread is also stopped if its main function returns.

time returns the current calendar time

```
#include <time.h>
time_t time(timer)
time_t *timer;
```

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 since 00:00:00 GMT on 1st January, 1970, according to the host system clock.

timer_after compare two transputer timer values

```
#include <timer.h>
int timer_after(t1, t2)
int t1, t2;
```

This function returns non-zero if timer value **t1** is *after* timer value **t2**, and zero otherwise.

timer_delay delay for some number of timer ticks

```
#include <timer.h>
timer_delay(d)
int d;
```

This function causes the current thread to wait for at least **d** ticks of the timer associated with the current thread's priority.

timer_now return the current timer value

```
#include <timer.h>
int timer_now()
```

This function returns the value of the timer associated with the current thread's priority.

timer_wait wait until current timer reaches some value

```
#include <timer.h>
timer_wait(t)
int t;
```

This function causes the current thread to wait until the value of the timer associated with the the priority of the current thread is at least **t**.

tolower convert char to lower case

```
int tolower(cval)
int cval;
```

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 convert char to upper case

```
int toupper(cval)
int cval;
```

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.

ungetc push character back into input stream

```
#include <stdio.h>
int ungetc(cval, stream)
FILE *stream;
int cval;
```

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 (q.v.) erases all memory of pushed back characters.

ungetc returns **EOF** if it can't push a character back.

unlink remove a file from the file system

```
int unlink (s)
char *s;
```

This function is identical to **remove** on this system, that is, the file identified by the string parameter **s** is deleted. If the file cannot be removed, the function returns −1.

write write on a file

```
int write(fildes, buffer, nbytes)
char *buffer;
int fildes, nbytes;
```

A file descriptor is the integer returned by a successful call on **open** or **creat**.

buffer is the address of **nbytes** contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Write returns −1 on error: bad descriptor, bad buffer address, bad count, or physical I/O errors.

17 Configuration language

The 3L configuration language is the language accepted by 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. For example, the flood-fill configurer accepts the barest descriptions of the user tasks; it needs no description of the physical network because that information will be discovered at load time.

17.1 Standard Syntactic Metalanguage

In a formal description of a computer language, it is often convenient to use a more precise language than English. This language-description language is referred to as a *metalanguage*. The metalanguage which will be used to describe the configuration language is that specified by British Standard 6154[7]. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory[8].

The BS6154 standard syntactic metalanguage is similar in concept to many other metalanguages, particularly those of the well-known Backus-Naur family. It therefore suffices to give a very brief informal description here of the main points of BS6154; for more detail, the standard itself should be consulted.

1 Terminal strings of the language — those not built up by rules of the language — are enclosed in quotation marks.

2 Non-terminal phrases are identified by names, which may consist of several words.

3 A sequence of items may be built up by connecting the components with commas.

4 Alternatives are separated by vertical bars ('|').

5 Optional sequences are enclosed in square brackets ('[' and ']').

6 Sequences which may be repeated zero or more times are enclosed in braces ('{' and '}').

7 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.

17.2 Configuration Language Syntax

To simplify the explanation of the configuration language, the formal definition which follows in subsections 17.2.2 onwards deals only with the higher level syntax of the language. At this level, we can deal with how the significant characters of the language are built up into tokens and statements. The lower level syntax deals with the way in which multiple input files are handled, with comments and with line continuation. This topic is treated informally in subsection 17.2.1.

The high level syntax given here has an additional simplification intended to make it 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 which 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, the two words here might be separated by a space character in the input file. When the two items are distinguishable in the input file without a space between them, then they may be abutted. This would be the case for the two items in the following example:

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

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

17.2.1 Low Level Syntax

The general form of a configuration language 'program' is designed to be as simple as possible to use.

The following example show 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:

- Case of letters is not significant to the configurer; in other words, 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.

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.

17.2.2 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, which are conveniently specified in units of kilobytes or megabytes. 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.

constant = *decimal constant* | *hex constant*;
hex constant = "`&`", *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, along with their values, expressed in decimal.

10	10
&10	16
10K	10 240
10M	10 485 760
1.6	1
1.6k	1 638

17.2.3 String Constants

The only circumstance in which a string constant is required in the configuration language is when an operating system file must be identified. Such string constants in the configuration language are simply enclosed in double quotes. No notation is available for including double quotes within the string; this is unnecessary as file names should not contain this character.

The trailing string quote may be omitted if the string is terminated by the end of the line.

string constant = *"", { any ASCII character other than newline or double quote }, [""];*

Some examples of valid string constants are as follows:

```
"string"
"fred.b4
```

17.2.4 Identifiers

Each object in the physical transputer 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 runs 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 last three examples would all be treated identically by the configurer, 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 which 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 which 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 ? host[0] root[0]
```

This statement declares a wire running from link number 0 on processor **host** to link number 0 on processor **root**. 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.

17.2.5 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 which follow.

statement = *processor statement*
| *wire statement*
| *task statement*
| *connect statement*
| *place statement*
| *bind 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.

17.2.6 PROCESSOR Statement

processor statement = "**PROCESSOR**",*new identifier*,{*processor attribute*};
processor attribute = "**TYPE**", "**=**", *processor type*;
processor type = "**PC**";

The PROCESSOR statement declares a physical processor. Every processor in the physical network must be declared, including the host processor from which the network is to be bootstrapped (normally the host computer). The configurer assumes that the processor named **host** is the host processor; thus, each configuration must contain a statement as follows:

```
processor host
```

Most processors declared in a configuration file will be declared so that user tasks can be placed on them by later statements. However, it is sometimes necessary to simply *describe* the tasks placed on a particular processor without causing them to be loaded into the processor. For example, the physical network may contain some processors which will already be executing tasks at the time the rest of the network is bootstrapped.

A trivial example of this case is the host processor itself. In the case of the host processor, the host will usually be executing the **iserver** program when the network is loaded, simply because that is the program which loads the rest of the network. It is necessary to be able to specify the **iserver** task to the configurer so that its ports can be connected to ports in user tasks, but without forcing the configurer to attempt to bootstrap the host computer. Similarly, some processors in the network might be set to bootstrap from ROM rather than from link; here, too, there is a need to describe the tasks running in those processors without attempting to bootstrap them.

A processor is declared to the configurer as having already been bootstrapped by means of the TYPE attribute taking the value **PC**. For example, a physical network containing one transputer and two host computers might be described as follows:

```
processor host
processor root_processor
processor other_host type=pc
```

Note that the default for the host is that it is **TYPE=PC** already. The default for all other processors is to be normal, bootable, transputer processors.

Every processor is assumed to be able to support any user task placed on it by the configuration file; specifically, there is no way to ask the configurer to check the memory requirements of tasks placed on the processor against the amount of physical memory available. Similarly, although certain tasks may not be able

to execute on particular types of processor (for example, a task making use of the floating point instructions found only on the T800 cannot execute on a T414), the configurer cannot check for this and the responsibility for ensuring a valid configuration is the user's.

Every processor in the network is assumed to have four INMOS links, numbered 0 to 3. These may be referred to (in the WIRE statement) by means of a link specifier construct, which consists of the processor identifier followed by the link number enclosed in square brackets:

link specifier = *processor identifier,* "**[**", *constant,* "**]**";

For example, link number 3 of the processor called **extra** would be specified as **extra[3]**.

17.2.7 WIRE Statement

wire statement = "**WIRE**", *new identifier, link specifier, link specifier;*

The WIRE statement declares a physical wire connecting links on two physical processors. Each wire supports two connections, one in either direction. The two link specifiers in the WIRE statement may therefore be interchanged without affecting the statement's meaning. For example, the following statements both 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]
```

17.2.8 TASK Statement

task statement	=	"**TASK**", *new identifier,* {*task attribute*};
task attribute	=	"**INS**", "**=**", *constant*
	|	"**OUTS**", "**=**", *constant*
	|	"**FILE**", "**=**", *task file specifier*
	|	"**OPT**", "**=**", *opt area*
	|	"**URGENT**"
	|	*memory area,* "**=**", *memory amount;*
opt area	=	*memory area* | "**CODE**";
memory area	=	"**STACK**"
	|	"**HEAP**"
	|	"**STATIC**"
	|	"**DATA**";
memory amount	=	*constant* | "**?**";
task file specifier	=	*identifier* | *string constant;*

The TASK statement declares a task, which may be either a user-supplied task or one of the standard tasks provided with the configurer. Each task statement 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

Each task declaration must include an INS attribute, which specifies 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 the number of input ports may be specified as 0.

OUTS Attribute

Each task declaration must include an OUTS attribute, which specifies 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 the number of output ports may be specified as 0.

FILE Attribute

This attribute specifies the file in which the memory image of the task is to be found. Task image files are produced by the **iboot** tool using the **C** option.

The FILE attribute is ignored for any processor which is declared as already having been bootstrapped, and may be omitted. This state is assumed for the host processor and for any processor for which the processor attribute **TYPE=PC** has been specified.

If the FILE attribute is omitted for a normal (bootable) processor, the configurer will scan the current directory and the directories specified in the environment variable **ISEARCH** for a file whose name is the same as the task's name, with the suffix '**.b4**'. The search stops at the first directory in which a file with the appropriate name is found.

If the FILE attribute is present, its argument is either a string constant, or a word with the same syntax as an identifier. In the former case, the string is the name of the file which will be opened, as in the following example:

```
task x file="mytask.b4" ...
```

If the identifier-like option is taken, the identifier given is used in a search through the **ISEARCH** path in the same way as the task's own identifier would have been

if the FILE attribute had been omitted:

```
task x file=mytask ...
```

Memory Size Attributes

The various memory size attributes specify the size of the various areas used as workspace for the task, as well as specifying which memory allocation strategy should be used.

The argument to one of the memory size attributes is an integer expressing the number of bytes of memory to be allocated to the area in question. Sizes smaller than 128 bytes 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. Only one task may request this treatment on any particular processor.

The *single-vector* allocation strategy is used if the DATA attribute appears. In this strategy, the task uses a single area of memory for all workspace requirements, whether stack, heap or static data. The stack and heap are allocated at opposite ends of this area, and grow towards each other. For example:

```
task x ... data=50k ...
```

The *double-vector* allocation strategy is used if the STACK and HEAP attributes appear (STATIC is available as a synonym for HEAP). In this strategy, the stack occupies a separate area of memory to all the other workspace used by a task. This can be useful when a task has a small stack requirement, as it can allow for the stack area to be placed into the transputer's on-chip memory using the task OPT attribute; this technique can produce large performance benefits. An example of double-vector allocation is as follows:

```
task x ... stack=1k heap=10k ...
```

The two allocation strategies are mutually exclusive. Thus, if the DATA size for the task is given, neither STACK nor HEAP should appear. If the two-vector allocation strategy is chosen, both STACK and HEAP must be specified. If no memory size attributes at all appear for a task, the default is the same as `DATA=?`; in other words, single-vector allocation of the rest of memory available on the processor.

OPT Attribute

This attribute specifies that the memory area given as its argument should be placed, if possible, into the transputer's on-chip memory area. The CODE specifier indicates the area of memory which will contain the executing code of the task; the other memory area specifiers have the same interpretation as for the memory size attributes.

If not all of the memory areas specified will fit into the on-chip memory, then some will be placed instead into the slower external memory, which is the default allocation for all memory areas. The order of precedence between memory areas in the same task is: stack, code, heap. In other words, if `OPT=STACK` and `OPT=CODE` are both specified, then the stack area is more likely to be placed in on-chip memory. No order of precedence is guaranteed between memory areas in different tasks.

It is possible for only part of a memory area to be placed in the on-chip RAM; this is useful in respect of the code area, where the modules which appeared first in the linker command line will have been placed at the start of the code area. If the most critical procedures are placed in the first module, then the likelihood of their being executed from on-chip memory will be increased.

The on-chip memory is quite small (2KB on the T414, 4KB on the T800), so the OPT attribute should be used sparingly to ensure that critical memory areas are not displaced into the slower external memory by less critical memory areas.

An example of a critical task with small stack and large data requirements might be as follows:

```
task t stack=1k heap=100k -
   opt=stack opt=code
```

URGENT Attribute

This attribute specifies that the task's initial thread is to be started at the urgent priority level. The default is that the task's initial thread is started at the not-urgent priority level. For example:

```
task x ... urgent ...
```

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).

17.2.9 CONNECT Statement

connect statement = *"**CONNECT**", new identifier,*
output port specifier, input port specifier;
output port specifier = *port specifier;*
input port specifier = *port specifier;*

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

```
connect ? iserver[0] filter[0]
connect ? filter[0] iserver[0]
```

Note that the order of the ports given in the CONNECT statement is significant, unlike the order of the links in the WIRE statement which CONNECT otherwise resembles.

17.2.10 PLACE Statement

place statement = *"**PLACE**", task identifier, processor identifier;*
processor identifier = *identifier;*
task identifier = *identifier;*

The PLACE statement determines which processor a particular task is to execute on; every task must be placed on some processor. A simple example of the use of this statement might be as follows:

```
place user_task root
place iserver host
```

17.2.11 BIND Statement

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

The BIND statement allows the contents of a port to be explicitly set to some literal value. Normally, ports are only bound by means of the CONNECT statement; ports left unbound are pointed at unique transputer channel words 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.

One application of the BIND statement is to give a task access to the transputer's external event mechanism. This appears as a channel word at address 80000020_{16}. Input port 5 of task `event_handler` could be initialised to point to this channel word as follows:

```
bind input event_handler[5] value=&80000020
```

Another application of the BIND statement is to pass an integer parameter to a user task. Here, the same input port as before is bound to the value 5:

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

Note that 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.

Appendices

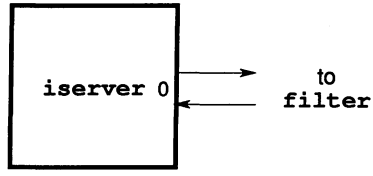
A Task data sheets

This chapter contains descriptions of the standard 'building block' tasks which are provided with Parallel C.

The description of each task starts with a diagram indicating the way in which the ports of the task should be connected to those of other tasks. Small digits inside the box representing the task are used to indicate port numbers corresponding to the connections visible outside the box.

This diagrammatic description is then backed up by a detailed description of the function of the task, along with examples of how a reference to the task might appear in a configuration file.

Data Sheet: `iserver`



The **iserver** task is used in configured applications to represent an **iserver** program executing on the host computer. It is therefore not provided in true task-image form.

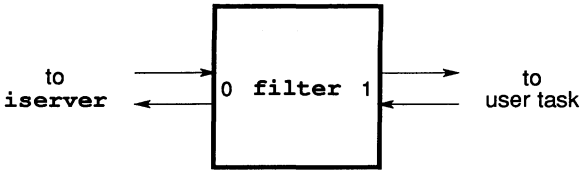
The **iserver** task should be described to the configurer as follows:

```
task iserver ins=1 outs=1
place iserver host
```

The **iserver** program (and therefore the **iserver** task) provides access to the host computer for tasks running in the transputer system, with which it communicates over its port pair 0.

The protocol used by the **iserver** is different to that used by the full C run-time library. Therefore the **iserver** must be attached to a **filter** task so that the **iserver** protocol is matched with the protocol used by user tasks.

Data Sheet: `filter`



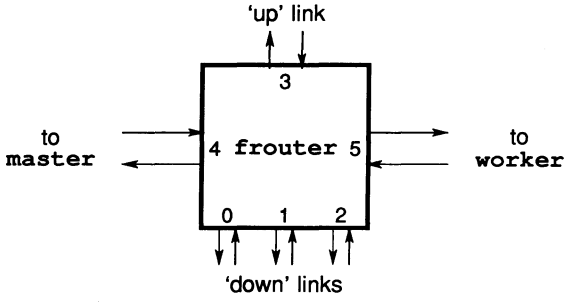
The **filter** task is used to convert between the protocol used by the **iserver** and the protocol used by the full run-time library for C. A **filter** task would be described in a configuration file as follows:

```
task filter ins=2 outs=2 data=10k
```

A **filter** task's port pair 0 communicates using the protocol expected by the **iserver** This is normally attached to an **iserver** task running on the host computer. Port pair 1 of a **filter** task communicates using the protocol expected by the full run-time library of C.

Thus, if a **filter** task is interposed between an **iserver** and a user task, they will be able to communicate normally although each is using a different protocol.

Data Sheet: `frouter`



The **`frouter`** task is used by the flood-filling configurer as the standard task which resides on each node of a flood-filled network and manages the flow of work packets and responses through the network.

The attributes used by the flood-filling configurer for the **`frouter`** task are as follows:

```
task router file=frouter ins=6 outs=6 -
             data=11k urgent
```

The following list summarises the way in which the **`frouter`** task is used by the flood-filling configurer:

0–2 Each of these pairs of 'down' ports are either set to zero by the loader, or are connected to the 'up' ports of nodes deeper in the network which were bootstrapped from this node. For each non-zero port pair in this range, the **`frouter`** task will start a pair of threads to carry packets to and from the subnetwork attached through that link.

3 If this node is *not* the root of the network, these 'up' ports are connected to a pair of 'down' ports of the router on the node which bootstrapped this node. In this case, the **`frouter`** task will read work packets and send responses to the booting node (and thus ultimately to the master task executing on the root node) through this pair of ports. If this node is the root of the network, these ports are set to zero by the loader and are ignored by the **`frouter`** task: port pair 4 (attached to the master task) will be used instead.

4 If this node is the root of the network, these ports are connected to the master task. In this case, the **`frouter`** task will read work packets and send responses to the master task through this port pair. Otherwise,

these ports are set to zero by the loader and the **frouter** task will use port pair 3 to reach the master task.

5 These ports are connected to the worker task executing on this node.

The standard **frouter** task uses two protocols in communicating with the tasks to which it is connected:

4–5 Port pairs connected directly to user tasks use the standard 'net' protocol described in section 8.2.

0–3 Port pairs connected to other routers through Inmos links use a variant of the 'net' protocol which is tolerant to the T414A problem with one-byte messages. In this variant, a two-byte message is actually transferred whenever the message header indicates that a one-byte message should follow.

Note that a communications task like **frouter** should normally be specified as having the **urgent** attribute. This prevents worker tasks in the network becoming idle because there is too little CPU time available elsewhere in the network for the router to operate.

B Harnesses and run-time libraries

In this release there are supplied a number of pre-compiled files that are used by the user when linking C programs (using **ilink**). These pre-compiled files are known as harnesses and are **always** required when linking a C program.

Also supplied are the full and reduced versions of the C run-time library files that have to be linked in with a C program as well.

B.1 Harnesses

There are two types of harness supplied with the release, one is used when linking a program for input to the configurers (**config** and **fconfig**) and the other is used when linking a program for input to the **iboot** tool.

The harness that is used when linking a program for input to the configurers is **taskharn.t*xx***. There are two versions of this harness, one for C programs compiled for the T414 (**taskharn.t4x**) and one for C programs compiled for the T800 (**taskharn.t8x**).

When a program has been linked using this type of harness it is then considered to be a *task* which can be placed on a node in a transputer network using the configurers supplied with this release. Before a linked task can be used by the configuration tools it must be converted to a format that is acceptable as input to the configurers. This conversion is performed using the **iboot** tool with its C option (see section 9.3).

The harness that is used when linking a program for input to the **iboot** tool is **mainent.c*xx***. There are also two versions of this harness, one for C programs compiled for the T414 (**mainent.c4x**) and one for C programs compiled for the T800 (**mainent.c4x**).

When a program has been linked with this type of harness it can then be considered as a compiled C program would be in a conventional non-parallel environment. In order to execute this type of C program it is necessary to make the linked program into an executable program. This operation is performed using the **iboot** tool (see section 9.1). Once the C program has been made executble it is then possible to load and execute it on a transputer board (using the host file server).

B.2 Run-time libraries

There are two versions of the run-time libraries supplied with this C release, a full run-time library and a reduced run-time library. The main difference between these two versions of the C run-time library is that the full C run-time library contains all the routines necessary to perform i/o to the host file system and terminal where as the reduced C run-time library does not contain these functions. The full C run-time library is called **crtl.lib** and the reduced C run-time library is called **sacrtl.lib**.

The full C run-time library is always used by C programs that are to be linked using the **mainent.cxx** harness. The full C run-time library is also used by C programs which are to be linked with the **taskharn.txx** harness and are to be configured so that they are connected to the host computer via the host file server (i.e. the **iserver**).

The reduced C run-time library is only used by C programs that are to to be linked using the **taskharn.txx** harness and are to be configured so that they are not connected to the host file server. (These programs will in general be executing on remote nodes in a transputer network with no access to host file and terminal services.)

B.2.1 Core maths run-time library

The C run-time library (full and reduced versions) share a common core of maths routines. Unlike the rest of the run-time library these routines are re-entrant and can be used in this way using the *occam 2 toolset*. That is, a single copy of the maths core of the C run-time library can be used by any number of C tasks executing on the same transputer when used called from an occam program.

The maths core is specified in the library build file **mrtl.lbb** and is referenced from the full and reduced library build files of the C run-time libraries (i.e. **crtl.lbb** and **sacrtl.lbb**). It is therefore possible to re-build the full and reduced C run-time libraries without the maths core in them so that only one copy of the maths core library is used when a number of C tasks are to be executed on the same transputer.

To build the maths core library and to rebuild the full and reduced versions of the C run-time libraries without the maths core it is first necessary to explode the full and reduced C run-time libraries as supplied into their component modules (using **ilibr** and its **X**) option). For example:

```
ilibr crtl.lib -x
ilibr sacrtl.lib -x
```

N.B. The reduced C run-time library contains a subset of the modules that appear in the full C run-time library so it does not matter if the librarian overwrites these common modules which have been exploded from the full C run-time library.

Once the standard (i.e. as supplied) full and reduced C run-time libraries have been exploded it will be necessary to modify the library build files for them. The only modification that is required is to remove the first line in the library build files that has the librarian command:

```
-f mrtl.lbb
```

To rebuild the full and reduced C run-time libraries as well as the core maths library the following sequence of commands should then be used:

```
ilibr -f mrtl.lbb -o mrtl.lib
ilibr -f crtl.lbb -o crtl.lib
ilibr -f sacrtl.lbb -o sacrtl.lib
```

It is now possible to use the library file **mrtl.lib** as a re-entrant library which need be linked in only once per transputer when more than one C task is to be executed on a single transputer.

To link a C task which requires the use of the core maths library it will necessary to perform serveral linking passes and to also use the linker's **-u** option.

For example, consider an occam program that calls two C tasks, both of which require routines from the core maths library where one C task uses the full C run-time library and the other uses the reduced C run-time library. The sequence of linking commands that will be required could be as follows:

ilink *h1 a1, a2, ...* **crtl.lib -o** *c1* **-u**
ilink *h2 b1, b2, ...* **sacrtl.lib -o** *c2* **-u**

Where *a1, a2, ...* are the object files for the first C task and *b1, b2, ...* are the object files for the second C task. *h1* and *h2* are the harness files that have to be linked with the C tasks and *c1* and *c2* are the output files corresponding to each partially linked C task.

To link these partially linked C tasks with the occam program that calls the C tasks the following linking command could be used:

ilink *o1, o2, ... c1, c2* **mrlt.lib** *ol1, ol2, ...* **-o** *o*

Where *o1, o2, ...* are the object files for the occam program and *ol1, ol2, ...* are the library files used by the occam program. *c1* and *c2* are the partially linked C tasks that are called by the occam program. *o* is the fully linked occam program with C tasks.

C Transputer instructions

This appendix provides a quick reference for the transputer instruction set as supported by Parallel C's **asm** statement. The syntax of the **asm** statement is covered in detail in section 14.5.

It is not anticipated that this appendix would be used as the sole reference for the transputer instruction set by a programmer unfamiliar with the transputer. For a detailed specification of each of the instructions available, refer to *'Transputer instruction set: a compiler writer's guide'* [12].

C.1 Pseudo-instructions

Pseudo-instructions are instructions to the assembler, rather than true transputer instructions. At present, only one pseudo-instruction is implemented, as follows:

byte This instruction takes as argument a list of constant values in the range 0 to 255. The assembler copies the literal bytes into the instruction stream.

C.2 Prefixing instructions

The transputer instruction set is built up from 16 *direct* instructions, each with a 4-bit argument field. The direct instructions include *prefix* instructions which augment the 4-bit field in a direct instruction which follows them by their own 4-bit argument field, effectively allowing the argument to be extended to 32 bits.

Normally, the assembler will compute the prefix instructions required for operand values greater than 4 bits automatically. However, you may wish to use explicit **pfix** and **nfix** instructions in conjunction with with the **byte** pseudo-instruction to synthesise special instruction sequences, for example for future transputer processors with additional instructions to those supported by Parallel C at present.

pfix prefix
nfix negative prefix

C.3 Direct instructions

The direct instructions form the core of the transputer instruction set. Each direct instruction has a single operand, normally an integer constant, which will be encoded in the instruction itself and, if it is larger than will fit into the 4-bit argument field of the direct instruction, into a series of **pfix** and **nfix** instructions as well.

The transputer architecture is based around a three-register *evaluation stack* and a single base register **Wreg**. The load and store 'local' instructions access a word in memory at a displacement from **Wreg** given by the operand value used. The displacement is scaled by the word size. The load and store 'non-local' instructions use the top evaluation stack register (**Areg**) as the base instead of **Wreg**, allowing computed base addresses to be used.

The operand of the **j**, **cj** and **call** instructions is interpreted as a byte displacement from the instruction pointer (program counter) register **Iptr**. **ldpi** is similar but takes its operand from **Areg**.

opr 'operate': the argument to this instruction is a code indicating a zero-operand *indirect* instruction to be executed. Most of the transputer instruction set is made up of these indirect instructions. Normally you would use the mnemonic for the specific indirect instruction which you require: the assembler will encode this as an **opr** instruction on your behalf. However, it is possible to use **opr** explicitly, for example to synthesise the instruction sequence for a new indirect instruction not supported by the T414 and T800 transputers.

ldc load constant

ldl load local word

stl store local word

ldlp load pointer to local word

adc add constant operand value to **Areg**

eqc test if **Areg** equals constant; **Areg** gets 1/0 result

j jump: the argument may be an identifier indicating a label for the jump to go to; the assembler will compute the displacement required.

cj conditional jump: as with **jump**, a label identifier may be used as argument to this instruction.

ldnl load non-local word

stnl store non-local word

ldnlp load pointer to non-local word

call call

ajw adjust workspace pointer **Wreg** by constant operand value (scaled by word length)

C.4 Operations

The instructions in this section are all *indirect* instructions built out of the **opr** instruction. None of these instructions take an argument; instead, they work solely with the transputer evaluation stack.

The arithmetic instructions take their operands from the top of the evaluation stack (**Areg**, **Breg**) and push the result value back on the stack in **Areg**.

rev	reverse top two stack elements
add	add
sub	subtract
mul	multiply
div	divide
rem	remainder
sum	sum
diff	difference
prod	product
and	bit-wise and
or	bit-wise inclusive or
xor	bit-wise exclusive or
not	bit-wise not
shl	shift left
shr	shift right
gt	greater than (1/0 result in **Areg**)
lend	loop end
bcnt	byte count
wcnt	word count
ldpi	load pointer to instruction (**Areg** is byte displacement from **Iptr**)
mint	minimum integer
bsub	byte subscript (**Areg = Areg + Breg**)
wsub	word subscript (**Areg = Areg + 4*Breg**)
move	move block of memory (src: **Creg** dest: **Breg** len: **Areg**)
in	input message
out	output message
lb	load byte
sb	store byte
outbyte	output byte
outword	output word

`gcall`	general call (swap `Areg`↔`Iptr`)
`gajw`	general adjust workspace
`ret`	return
`startp`	start process
`endp`	end process
`runp`	run process
`stopp`	stop process
`ldpri`	load current priority
`ldtimer`	load timer
`tin`	timer input
`alt`	alt start
`altwt`	alt wait
`altend`	alt end
`talt`	timer alt start
`taltwt`	timer alt wait
`enbs`	enable skip
`diss`	disable skip
`enbc`	enable channel
`disc`	disable channel
`enbt`	enable timer
`dist`	disable timer
`csub0`	check subscript from 0
`ccnt1`	check count from 1
`testerr`	test error false and clear
`stoperr`	stop on error
`seterr`	set error
`xword`	extend to word
`cword`	check word
`xdble`	extend to double
`csngl`	check single
`ladd`	long add
`lsub`	long subtract

`lsum`	long sum
`ldiff`	long difference
`lmul`	long multiply
`ldiv`	long divide
`lshl`	long shift left
`lshr`	long shift right
`norm`	normalise
`resetch`	reset channel
`testpranal`	test processor analysing
`sthf`	store high priority front pointer
`stlf`	store high priority back pointer
`sttimer`	store timer
`sthb`	store high priority back pointer
`stlb`	store low priority back pointer
`saveh`	save high priority queue registers
`savel`	save low priority queue registers
`clrhalterr`	clear halt-on-error
`sethalterr`	set halt-on-error
`testhalterr`	test halt-on-error
`fmul`	fractional multiply

C.5 T414-only instructions

The indirect instructions in this section may only be executed on T414 processors, although you may use them in **asm** statements even when compiling for a different processor.

`unpacksn`	unpack single-length floating-point number
`roundsn`	round single-length floating-point number
`postnormsn`	post-normalise correction of single-length floating-point number
`ldinf`	load single-length infinity
`cflerr`	check single-length floating-point infinity or not-a-number

C.6 T800-only instructions

The instructions in this section may only be executed on T800 processors, although you may use them in **asm** statements even when compiling for a different processor.

C.6.1 Floating-point instructions

The indirect instructions in this section provide access to the T800's built-in floating-point processor. Note that the instructions beginning with '**fpu**...' are doubly indirect: they are accessed by loading an *entry code* constant with a **ldc** instruction, then executing an **fpentry** instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the **fpentry** instruction is also available.

The floating point load and store instructions use the *integer* **Areg** as a pointer to the operand location.

fpentry	floating point unit entry: used to synthesise the '**fpu**...' instructions.
fpdup	floating duplicate
fprev	floating reverse
fpldnlsn	floating load non-local single
fpldnldb	floating load non-local double
fpldnlsni	floating load non-local indexed single
fpldnldbi	floating load non-local indexed double
fpstnlsn	floating store non-local single
fpstnldb	floating store non-local double
fpurn	set rounding mode to round nearest
fpurz	set rounding mode to round zero
fpurp	set rounding mode to round positive
fpurm	set rounding mode to round minus
fpadd	floating-point add
fpsub	floating-point subtract
fpmul	floating-point multiply
fpdiv	floating-point divide
fpusqrtfirst	floating-point square root first step
fpusqrtstep	floating-point square root step
fpusqrtlast	floating-point square root end
fpremfirst	floating-point remainder first step
fpremstep	floating-point remainder iteration step
fpldzerosn	load zero single
fpldzerodb	fload zero double
fpmulby2	multiply by 2.0
fpudivby2	divide by 2.0
fpuexpinc32	multiply by 2^{32}
fpuexpdec32	divide by 2^{32}
fpuabs	floating-point absolute

`fpldnladdsn`	floating load non-local and add single
`fpldnladddb`	floating load non-local and add double
`fpldnlmulsn`	floating load non-local and multiply single
`fpldnlmuldb`	floating load non-local and multiply double
`fpchkerr`	check floating error
`fptesterr`	test floating error false and clear
`fpuseterr`	set floating error
`fpuclrerr`	clear floating error
`fpgt`	floating point greater than
`fpeq`	floating point equality
`fpordered`	floating point orderability
`fpnan`	floating point not-a-number
`fpnotfinite`	floating point finite
`fpur32tor64`	convert single to double
`fpur64tor32`	convert double to single
`fpint`	round to floating integer
`fpstnli32`	store non-local 32-bit integer
`fpuchki32`	check in range of 32-bit integer
`fpuchki64`	check in range of 64-bit integer
`fprtoi32`	convert floating to 32-bit integer
`fpi32tor32`	convert 32-bit integer to 32-bit real
`fpi32tor64`	convert 32-bit integer to 64-bit real
`fpb32tor64`	convert 32-bit unsigned integer to 64-bit real
`fpunoround`	convert 64-bit real to 32-bit real without rounding

C.6.2 Other T800-only instructions

The indirect instructions in this section supplement the T414's integer instruction set.

`dup`	duplicate top of stack
`move2dinit`	initialise data for 2-dimensional block move
`move2dall`	2-dimensional block copy
`move2dnonzero`	2-dimensional block copy non-zero bytes
`move2dzero`	2-dimensional block copy zero bytes
`crcword`	calculate Cyclic Redundancy Check (CRC) on word
`crcbyte`	calculate CRC on byte
`bitcnt`	count the number of bits set in a word
`bitrevword`	reverse bits in a word
`bitrevnbits`	reverse bottom n bits in a word
`wsubdb`	form double-word subscript

D Conventions and defaults

All tools in the toolset, and all implementations of the toolset, use a common set of conventions and defaults.

The toolset conforms to conventions in the following areas:

- command line syntax and options
- error handling and message format
- file naming
- file location.

D.1 Command line conventions

Syntax

All tools in the toolset conform to the following syntax conventions:

- Options must be prefixed by the option prefix character. The option prefix character is '–' for UNIX based toolsets, and '/' for VAX VMS and DOS based toolsets.
- Options may occur anywhere on the command line and case is ignored.
- If an option takes more than one parameter the parameters must be enclosed in parentheses `()`, and separated by commas.

Common options

All tools in the toolset conform to the following conventions for command line options:

- All tools provide help information if called with no options.
- The `-I` option, where supported, displays information about what the tool is doing.
- The `-F` option, where supported, is used to specify an input filename. If no name is given then input is taken from the host system standard input (normally the keyboard).

- The `-L` option, where supported, loads the program without invoking the tool or running the program. This can be used to load a program onto a transputer board without running it, or to test for the existence of a particular tool.

- The `-O` option, where supported, is used to specify an output filename. If no name is give then output is sent to the host system standard output (normally the screen).

D.2 Filename conventions

The toolset encourages the use of conventions in file naming, and especially in the area of file extensions. The relationship of tools to each other in the various stages of program building and compilation, relies on these conventions, and it is recommended that you use them wherever possible.

Filename conventions are encouraged for three reasons. Firstly, it enables filenames to be used in a host independent manner. Secondly, it enables file extensions to be omitted from filenames in many commands, because defaults can be assumed. This simplifies the use of these commands.

Filenames

Filenames should not contain the characters: dot `.`, colon `:`, semi-colon `;`, square brackets `[]`, round brackets `()`, forward slash `/`, backslash \, exclamation mark `!`, equals =.

Where the host operating system allows logical names to be used in place of filenames, such as with VMS, the toolset allows logical names to be used, but the name must be followed by a dot (.). This prevents the tool from adding an extension, which would generate a host filing system error.

File extensions

File extensions used by the software, and the tools to which they relate, are given in table D.1.

Extension	Tool	File Type	Description
`.b4`	`config` `fconfig`	Input	Configurer input files. Input files created by the `iboot` tool using the tool's 'C' and 'O' options.
`.bt`	`config` `fconfig`	Output	Configurer output file.
`.bxx`	`iboot`	Output	Bootable code for single transputer.
`.cxx`	`ilink`	Output	Linked code files.
`.dxx`	`iboot`	Output	Bootstrap tool code description.
`.h`	–	–	Declarations of constants etc.
`.lbb`	`ilibr`	Input	Library build file.
`.lib`	`ilibr`	Output	Library file.
`.lxx`	`ilink`	Input	Linker command files.
`.mxx`	`ilink`	Output	Module map.
`.c`	`tc` / `t4c` / `t8c`	Input	C source.
`.cfg`	`config` `fconfig`	Input	Network configuration file.
`.sxx`	`ilink`	Output	Symbol table produced by the linker.
`.bin`	`tc` / `t4c` / `t8c`	Output	Code files produced by compiler.

N.B. for extensions `.bxx`, `.cxx`, etc., the value of *x* depends on the transputer type (**4** for T414 and **8** for T800).

Table D.1 File extensions

In file extensions composed of a letter and two additional characters, for example `.bxx`, the second character of the extension indicates the transputer type.

If an extension is not specified, some tools assume an appropriate extension. For example, the C compiler `t4c` (or `t8c`) assumes the extension `.c` on the input file, and the configurer tool `config` assumes a `.b4` extension on its input files.

Other tools, such as `ilink`, cannot make assumptions about the input file to use, and require the extension to be explicitly stated.

D.3 File location conventions

The tools locate files by searching a specified directory *path* on the host system. The path is specified on the PC using the environment variable **ISEARCH**, and on VAX systems running VMS, by a sequence of logical names.

The tools conform to the following rules for locating files:

1 If the filename contains a directory specification then the filename is used as given. Relative directory names are treated as relative to the directory in which the tool was invoked.

2 If no directory is specified the directory in which the tool was invoked is searched.

3 If the file is not present in the current directory, the path specified by the environment variable (or logical name) **ISEARCH** is searched. If there are several files of the same name on this path, the first occurrence is used.

4 If the file is not found using the above rules, then the file is assumed to be absent, and an error is generated by the tool.

If no search path has been set up then only rule 1 applies.

D.4 Search paths on the IBM PC and SUN3

When using the environment variable **ISEARCH** on a PC running DOS and the SUN3 running SunOS, directories to be searched are specified as a list of directory paths. Directories are searched in the order that they appear in the list.

Within the list, directory paths must be followed by the appropriate directory separator character ('\' for DOS and '/' for SunOS), and entries in the list must be separated by a space or a semi-colon.

D.5 Search paths on VMS systems

The symbol **ISEARCH** has a list of logical names as its argument, which by convention will be **ISEARCH_1**, **ISEARCH_2**, etc., separated by spaces.

You should set up the logical names to point to the directories that you require. They will be searched in the order that they are specified when the symbol

`ISEARCH` is set up. For example:

```
$ ISEARCH :=="ISEARCH_1: ISEARCH_2:"
```

E ASCII code chart

	0x0*x*	0x1*x*	0x2*x*	0x3*x*	0x4*x*	0x5*x*	0x6*x*	0x7*x*
0x*x*0	NUL	DLE		0	@	P	`	p
0x*x*1	SOH	DC1	!	1	A	Q	a	q
0x*x*2	STX	DC2	"	2	B	R	b	r
0x*x*3	ETX	DC3	#	3	C	S	c	s
0x*x*4	EOT	DC4	$	4	D	T	d	t
0x*x*5	ENQ	NAK	%	5	E	U	e	u
0x*x*6	ACK	SYN	&	6	F	V	f	v
0x*x*7	BEL	ETB	'	7	G	W	g	w
0x*x*8	BS	CAN	(	8	H	X	h	x
0x*x*9	HT	EM	)	9	I	Y	i	y
0x*x*A	LF	SUB	*	:	J	Z	j	z
0x*x*B	VT	ESC	+	;	K	[	k	{
0x*x*C	FF	FS	,	<	L	\	l	|
0x*x*D	CR	GS	-	=	M	]	m	}
0x*x*E	SO	RS	.	>	N	^	n	~
0x*x*F	SI	US	/	?	O	_	o	DEL

Bibliography

[1] *The C Programming Language.* Brian W. Kernighan and Dennis M. Ritchie. Prentice-Hall, 1978. ISBN 0-13-110163-3.

[2] *Disk Operating System Version 3.10 Reference.* International Business Machines, February 1985.

[3] *Microsoft MS-DOS User's Reference.* Microsoft Corporation, 1986. Document Number 410630013-320-R03-0686.

[4] *Disk Operating System Version 3.00 Technical Reference.* International Business Machines, May 1984.

[5] A. M. Lister, *Fundamentals of Operating Systems.* Macmillan Press, 1979. ISBN 0-333-27287-0.

[6] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation.* Prentice-Hall, 1987. ISBN 0-13-637331-3.

[7] *British Standard BS6154:1982: Method of Defining Syntactic Metalanguage.* British Standards Institution, 1981. ISBN 0-580-12530-0.

[8] R. S. Scowen. *An Introduction and Handbook for the Standard Syntactic Metalanguage.* National Physical Laboratory Report DITC 19/83, February 1983.

[9] *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic.* Institute of Electrical and Electronics Engineers, 1985.

[10] *IMS T414 transputer: Engineering data.* Inmos Ltd., June 1987.

[11] *IMS T800 transputer: Preliminary Data.* Inmos Ltd., April 1987.

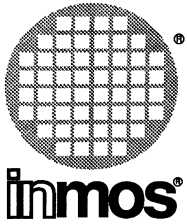
[12] *The transputer instruction set: a compiler writers' guide.* Inmos Ltd., February 1987. Publication number 72 TRN 119 01.

[13] Roger Shepherd. *Technical Note 1: Extraordinary use of transputer links.* Inmos Ltd., November 1986.

[14] Stephen Ghee. *Technical Note 11: IMS B004 IBM PC add-in board.* Inmos Ltd., February 1987.

[15] Michael Rygol and Trevor Watson. *Technical Note 18: Connecting Inmos Links.* Inmos Ltd., April 1987.

Index



INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
U.K.
Telephone (0454) 616616
TLX 444723

INMOS SARL
Immeuble Monaco
7 rue Le Corbusier
SILIC 219
94518 Rungis Cedex
France
Telephone (1) 46.87.22.01
TLX 201222

INMOS GmbH
Danziger Strasse 2
8057 Eching
West Germany
Telephone (089) 319 10 28
TLX 522645

INMOS Corporation
P.O. Box 16000
Colorado Springs
Colorado 80935
U.S.A.
Telephone (719) 630 4000
TLX (Easy Link) 62944936

INMOS Japan K.K.
4th Floor No 1 Kowa Bldg
11-41 Akasaka 1-chome
Minato-ku
Tokyo 107
Japan
Telephone 03-505-2840
TLX J29507 TEI JPN