

# Contents

<b>Introduction</b>	<b>xv</b>
Intended Audience . . . . .	xv
The C Language . . . . .	xv
Hardware Assumptions . . . . .	xvi
Document Structure . . . . .	xvii
Further Reading . . . . .	xviii
Conventions . . . . .	xix
Text Conventions . . . . .	xix
Installation Directory . . . . .	xix
<b>1 Installing the Compiler</b>	<b>1</b>
1.1 Installation Directory . . . . .	1
1.2 Installing the Software . . . . .	2
1.3 The Search Path . . . . .	3
1.4 Environmental variable 3LCC_INC . . . . .	4
<b>2 Confidence Testing</b>	<b>7</b>
<b>3 Developing Sequential Programs</b>	<b>11</b>
3.1 Editing . . . . .	12
3.2 Compiling . . . . .	12
3.3 Linking . . . . .	13
3.3.1 Linking More than One Object File . . . . .	14
3.3.2 Indirect Files . . . . .	15
3.3.3 Calling the Linker Directly . . . . .	16
3.3.4 Libraries . . . . .	17

3.4	Running . . . . .	19
3.4.1	Using C Programs as MS-DOS Commands . . .	20
3.4.2	Command-Line Arguments . . . . .	21
3.4.3	I/O Redirection and Piping . . . . .	22
3.5	Memory Use . . . . .	23
3.5.1	Default Memory Mapping . . . . .	24
3.5.2	Alternative Memory Mapping . . . . .	25
3.5.3	Limit on Program Memory . . . . .	25
<b>4</b>	<b>Introduction to Parallel C</b>	<b>27</b>
4.1	Abstract Model . . . . .	27
4.2	Hardware Realisation . . . . .	29
4.3	Software Model . . . . .	30
4.4	Simultaneous Input . . . . .	32
4.5	Parallel Execution Threads . . . . .	33
4.6	Configuring an Application . . . . .	33
4.7	Processor Farms . . . . .	34
<b>5</b>	<b>Developing Parallel Programs</b>	<b>37</b>
5.1	Configuring One User Task . . . . .	38
5.1.1	Hardware Configuration . . . . .	40
5.1.2	Software Configuration . . . . .	41
5.1.3	Building the Application . . . . .	43
5.2	More than One User Task . . . . .	46
5.2.1	Inter-Task Communication Functions . . . . .	47
5.3	Building Multi-Task Systems . . . . .	51
5.4	Multi-Transputer Systems . . . . .	53
5.5	Simultaneous Input . . . . .	54
5.6	Multi-Threaded Tasks . . . . .	57
5.6.1	Creating Threads . . . . .	57
5.6.2	Threads versus Tasks . . . . .	60
5.7	Debugging . . . . .	63
5.8	Estimating Memory Requirements . . . . .	65
<b>6</b>	<b>Global Input/Output</b>	<b>67</b>
6.1	One Transputer . . . . .	67
6.2	More than One Transputer . . . . .	70

6.3	More than One Multiplexer . . . . .	71
6.4	Limits . . . . .	71
6.5	Termination of an Application . . . . .	72
<b>7</b>	<b>Processor Farms</b>	<b>77</b>
7.1	The Worker Task . . . . .	79
7.2	The Master Task . . . . .	79
7.3	The <code>net</code> Package . . . . .	80
7.3.1	Functions <code>net_send</code> and <code>net_receive</code> . . . . .	81
7.3.2	The <code>net_broadcast</code> function . . . . .	82
7.4	Building the Application . . . . .	82
7.4.1	Configuration File . . . . .	83
7.5	Running the Example . . . . .	84
7.6	Heterogeneous Networks . . . . .	86
<b>8</b>	<b>Developing T2 Programs</b>	<b>89</b>
8.1	Compiling . . . . .	89
8.2	The Compiler in T2 Mode . . . . .	90
8.2.1	Language Restrictions . . . . .	90
8.2.2	Pre-defined Macros . . . . .	91
8.2.3	Data-type Representations . . . . .	92
8.2.4	Compiler Error Messages . . . . .	93
8.3	Linking T2 Tasks . . . . .	93
8.4	Linker Support for the T2 . . . . .	94
8.4.1	Linker Command Switches . . . . .	94
8.4.2	The Bootstrap . . . . .	99
8.5	The Run-Time Library . . . . .	99
8.5.1	Functions Defined in <code>alt.h</code> . . . . .	100
8.5.2	Functions Defined in <code>chan.h</code> . . . . .	100
8.5.3	Functions Defined in <code>chanio.h</code> . . . . .	100
8.5.4	Functions Defined in <code>ctype.h</code> . . . . .	100
8.5.5	Functions Defined in <code>locale.h</code> . . . . .	101
8.5.6	Functions Defined in <code>par.h</code> . . . . .	101
8.5.7	Functions Defined in <code>sema.h</code> . . . . .	101
8.5.8	Functions Defined in <code>setjmp.h</code> . . . . .	101
8.5.9	Functions Defined in <code>signal.h</code> . . . . .	101
8.5.10	Functions Defined in <code>stdlib.h</code> . . . . .	101

8.5.11	Functions Defined in <code>string.h</code> . . . . .	102
8.5.12	Functions Defined in <code>thread.h</code> . . . . .	102
8.5.13	Functions Defined in <code>timer.h</code> . . . . .	102
8.6	Running T2 Programs . . . . .	102
8.6.1	Using the Configurer to Boot a T2 . . . . .	103
8.6.2	Piping Code into a T2 . . . . .	104
8.7	Parameters to Main . . . . .	106
<b>Introduction</b> . . . . .		<b>107</b>
	Overview . . . . .	107
	Standard Syntactic Metalanguage . . . . .	108
<b>9</b>	<b>C Compiler Reference</b> . . . . .	<b>109</b>
9.1	The C Language . . . . .	109
9.1.1	ANSI Features . . . . .	110
9.1.2	Special Features . . . . .	119
9.1.3	System-dependent Features . . . . .	120
9.2	The C <code>main</code> Function . . . . .	121
9.3	Running the Compiler . . . . .	121
9.4	Compiler Switches . . . . .	122
9.4.1	Default Switches . . . . .	123
9.4.2	Controlling Output Files . . . . .	124
9.4.3	Controlling Object Code . . . . .	126
9.4.4	Controlling Code Patch Sizes . . . . .	129
9.4.5	Controlling Debugging . . . . .	131
9.4.6	Controlling <code>#include</code> Processing . . . . .	132
9.4.7	Macro Definitions . . . . .	132
9.4.8	Information from the Compiler . . . . .	133
9.5	Predefined Macros . . . . .	135
9.6	Handling of <code>#include</code> Files . . . . .	135
9.7	Assembly Language . . . . .	137
9.7.1	When to Use Assembly Language . . . . .	137
9.7.2	Assembly Language Syntax . . . . .	138
9.7.3	Literal Operands . . . . .	139
9.7.4	Variables as Operands . . . . .	139
9.7.5	Accessing Complex Structures . . . . .	142
9.7.6	Labels and Jumps . . . . .	144

9.7.7	Literal Machine Code . . . . .	146
9.7.8	Errors . . . . .	146
9.8	Data-type Representations . . . . .	147
9.8.1	Integral Data Types . . . . .	147
9.8.2	Pointer Types . . . . .	148
9.8.3	Floating Types . . . . .	148
9.8.4	Alignment and Complex Types . . . . .	150
9.9	Compiler Error Messages . . . . .	151
9.9.1	Compiler Error Message Format . . . . .	152
9.9.2	Fixing Errors Detected by the Compiler . . . .	154
9.9.3	List of Error Messages . . . . .	156
9.9.4	Errors in Assembler Language . . . . .	198
<b>10</b>	<b>The C Run-Time Library</b>	<b>203</b>
10.1	Introduction . . . . .	203
10.1.1	Purpose of the Run-Time Library . . . . .	203
10.1.2	Versions of the Run-Time Library . . . . .	204
10.1.3	Conventions . . . . .	205
10.1.4	Header Files . . . . .	206
10.1.5	Errors <code>&lt;errno.h&gt;</code> . . . . .	207
10.1.6	Limits <code>&lt;float.h&gt;</code> and <code>&lt;limits.h&gt;</code> . . . . .	208
10.1.7	Common Definitions <code>&lt;stddef.h&gt;</code> . . . . .	208
10.2	Alt Package <code>&lt;alt.h&gt;</code> . . . . .	208
10.3	Diagnostics <code>&lt;assert.h&gt;</code> . . . . .	209
10.4	Neighbouring Transputers <code>&lt;boot.h&gt;</code> . . . . .	209
10.5	Channels <code>&lt;chan.h&gt;</code> . . . . .	209
10.6	Character Handling <code>&lt;ctype.h&gt;</code> . . . . .	211
10.6.1	Character Testing Functions . . . . .	211
10.6.2	Character Mapping Functions . . . . .	212
10.7	Accessing DOS Functions <code>&lt;dos.h&gt;</code> . . . . .	212
10.8	Localisation <code>&lt;locale.h&gt;</code> . . . . .	217
10.9	Mathematics <code>&lt;math.h&gt;</code> . . . . .	217
10.9.1	Treatment of Error Conditions . . . . .	218
10.9.2	Trigonometric Functions . . . . .	218
10.9.3	Hyperbolic Functions . . . . .	218
10.9.4	Exponential and Logarithmic Functions . . . .	218
10.9.5	Power Functions . . . . .	219

10.9.6 Nearest Integer, Absolute Value and Remain- der Functions . . . . .	219
10.10 Processor Farm Communications <code>&lt;net.h&gt;</code> . . . . .	220
10.11 Synchronising Access to Run-Time Library <code>&lt;par.h&gt;</code> . . . . .	220
10.12 Semaphores <code>&lt;sema.h&gt;</code> . . . . .	221
10.13 Emulating the <code>filter</code> Task <code>&lt;serv.h&gt;</code> . . . . .	221
10.14 Nonlocal Jumps <code>&lt;setjmp.h&gt;</code> . . . . .	221
10.15 Signal Handling <code>&lt;signal.h&gt;</code> . . . . .	222
10.16 Variable Arguments <code>&lt;stdarg.h&gt;</code> . . . . .	222
10.17 Input/Output <code>&lt;stdio.h&gt;</code> . . . . .	223
10.17.1 Stream I/O . . . . .	225
10.17.2 Binary I/O . . . . .	226
10.17.3 Text I/O . . . . .	226
10.17.4 Operations on Files . . . . .	227
10.17.5 File Access Functions . . . . .	227
10.17.6 Formatted Input/Output Functions . . . . .	227
10.17.7 Character Input/Output Functions . . . . .	228
10.17.8 Direct Input/Output Functions . . . . .	229
10.17.9 File Positioning Functions . . . . .	229
10.17.10 Error Handling Functions . . . . .	230
10.18 General Utilities <code>&lt;stdlib.h&gt;</code> . . . . .	230
10.18.1 String Conversion Functions . . . . .	230
10.18.2 Pseudo-Random Sequence Generation Functions	230
10.18.3 Memory Management Functions . . . . .	231
10.18.4 Communication with the Environment . . . . .	231
10.18.5 Searching and Sorting Utilities . . . . .	232
10.18.6 Integer Arithmetic Functions . . . . .	232
10.18.7 Multibyte Character Functions . . . . .	232
10.18.8 Multibyte String Functions . . . . .	232
10.19 String Handling <code>&lt;string.h&gt;</code> . . . . .	233
10.19.1 Copying Functions . . . . .	233
10.19.2 Concatenation Functions . . . . .	233
10.19.3 Comparison Functions . . . . .	233
10.19.4 Search Functions . . . . .	234
10.19.5 Miscellaneous Functions . . . . .	234
10.20 Threads <code>&lt;thread.h&gt;</code> . . . . .	235

10.21	Date and Time <code>&lt;time.h&gt;</code> . . . . .	235
10.22	Transputer Timers <code>&lt;timer.h&gt;</code> . . . . .	236
<b>11</b>	<b>Alphabetic List of Run-time Library Entries</b>	<b>237</b>
<b>12</b>	<b>The Linker</b>	<b>333</b>
12.1	Command Line . . . . .	333
12.2	File Name Conventions . . . . .	334
12.3	The Output File . . . . .	335
12.4	Indirect Files . . . . .	335
12.5	Libraries . . . . .	336
12.6	The Executable Image . . . . .	338
12.7	Map Files . . . . .	340
12.8	T2 Support . . . . .	340
12.8.1	Switch <code>/Msize</code> . . . . .	341
12.8.2	Switch <code>/Asize</code> . . . . .	341
12.8.3	Switches <code>/FC</code> , <code>/FA</code> , <code>/FS</code> , and <code>/FH</code> . . . . .	342
12.8.4	Modified <code>/F</code> Switches . . . . .	344
12.8.5	Switch <code>/Rsize</code> . . . . .	345
12.9	Debug Tables . . . . .	345
12.10	Summary of Switches . . . . .	345
12.11	Using Batch Files . . . . .	348
12.12	Duplicate Definitions . . . . .	349
12.13	Messages . . . . .	350
<b>13</b>	<b>The <code>mempatch</code> Utility</b>	<b>367</b>
13.1	Identifying <code>mempatch</code> . . . . .	368
13.2	Invoking <code>mempatch</code> . . . . .	369
13.3	Re-invoking <code>mempatch</code> . . . . .	369
<b>14</b>	<b>The <code>decode</code> Utility</b>	<b>371</b>
14.1	Usage . . . . .	371
14.1.1	Compilation for the Decoder . . . . .	371
14.1.2	Running the Decoder . . . . .	372
14.2	Features of the <code>decode</code> Program . . . . .	372
14.3	Other Languages . . . . .	373

<b>15 The worm Utility</b>	<b>375</b>
15.1 Notes . . . . .	376
<b>16 The tnm Utility</b>	<b>379</b>
<b>17 The tunlib Utility</b>	<b>381</b>
<b>18 Configuration Language Reference</b>	<b>383</b>
18.1 Standard Syntactic Metalanguage . . . . .	383
18.2 Configuration Language Syntax . . . . .	384
18.2.1 Low Level Syntax . . . . .	385
18.2.2 Numeric Constants . . . . .	387
18.2.3 String Constants . . . . .	388
18.2.4 Identifiers . . . . .	389
18.2.5 Statements . . . . .	391
18.2.6 PROCESSOR Statement . . . . .	391
18.2.7 WIRE Statement . . . . .	397
18.2.8 TASK Statement . . . . .	397
18.2.9 CONNECT Statement . . . . .	403
18.2.10 PLACE Statement . . . . .	404
18.2.11 BIND Statement . . . . .	405
<b>19 Flood-Fill Configurer Reference</b>	<b>407</b>
19.1 User Task Protocol . . . . .	407
19.1.1 Master Task's Ports . . . . .	408
19.1.2 Worker Task's Ports . . . . .	408
19.2 Packet Format . . . . .	408
<b>20 Task Data Sheets</b>	<b>411</b>
 <b>Appendices</b>	 <b>425</b>
<b>A Distribution Kit</b>	<b>425</b>
A.1 Directory \tc2v2 . . . . .	425
A.2 Directory \tc2v2\examples . . . . .	427
 <b>B Compatibility with T414A and T800A</b>	 <b>429</b>



B.1	Problems with T414A . . . . .	429
B.1.1	Restriction on Message Lengths . . . . .	430
B.1.2	Problems with Timers . . . . .	430
B.2	Problems with T800A . . . . .	431
B.2.1	Floating-Point Conversion Problems . . . . .	431
B.2.2	Instruction Decode Problems . . . . .	431
<b>C</b>	<b>Building a Network</b>	<b>433</b>
C.1	Network Principles . . . . .	433
C.2	Network Requirements . . . . .	434
C.2.1	Requirements for Links . . . . .	434
C.2.2	Requirements for System Services . . . . .	435
C.3	Connecting a Network . . . . .	436
<b>D</b>	<b>Summary of Option Switches</b>	<b>439</b>
D.1	Compiler Switches . . . . .	439
D.2	Linker Switches . . . . .	441
D.3	<code>afserver</code> Switches . . . . .	443
D.4	General Purpose Configurer Switches . . . . .	445
<b>E</b>	<b>Transputer Instructions</b>	<b>447</b>
E.1	Pseudo-Instructions . . . . .	447
E.2	Prefixing Instructions . . . . .	448
E.3	Direct Instructions . . . . .	448
E.4	Operations . . . . .	450
E.5	T4-only Instructions . . . . .	452
E.6	T8-only Instructions . . . . .	453
E.6.1	Floating Point Instructions . . . . .	453
E.6.2	Other T8-only Instructions . . . . .	455
<b>F</b>	<b>Compatibility Functions</b>	<b>457</b>
F.1	Introduction . . . . .	457
F.1.1	ASCII Control Codes <code>&lt;ascii.h&gt;</code> . . . . .	458
F.1.2	Channel Communications <code>&lt;chanio.h&gt;</code> . . . . .	458
F.1.3	Variable Arguments <code>&lt;varargs.h&gt;</code> . . . . .	458
F.2	Low-Level I/O . . . . .	459
F.3	Alphabetic List of Compatibility Functions . . . . .	460

<b>G Mandelbrot Program Listings</b>	<b>473</b>
G.1 Mandelbrot Example Master Task . . . . .	473
G.2 Mandelbrot Example Worker Task . . . . .	479
G.3 Header File . . . . .	481
G.4 Configuration File . . . . .	481
<b>H ASCII Code Chart</b>	<b>483</b>
<b>Bibliography</b>	<b>485</b>
<b>Index</b>	<b>487</b>

# Introduction

## Intended Audience

This *User Guide* accompanies 3L's Parallel C product. It 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.

## The C Language

There are two main dialects of the C language in common use: these are often referred to as “K&R C” and “ANSI C”.

**K&R C** This older dialect of C is defined—fairly informally—by *The C Programming Language, First Edition*[1], by Brian W. Kernighan and Dennis M. Ritchie, the original authors of the language.

**ANSI C** This is defined, in ANS X3.159-1989[3], as the American national standard for the C language. At the time of writing, the same definition was expected to be adopted as an international standard.

The dialect of C accepted by the 3L Parallel C was originally based on K&R C. However, it has been extended by adding most of the features of ANSI C, including, for example, function prototypes and enumerated types. Details of Parallel C's ANSI extensions may be found in section 9.1.

In addition, the run-time library includes nearly all of the features of the ANSI run-time library. Traditional features have been retained as well, for compatibility with other compilers. To this have been added functions providing control of the transputer's special features, such as channel communications, concurrent execution threads, and so on.

## Hardware Assumptions

Parallel C can be used with a large variety of development and target transputer systems.

The compiler itself and all the supporting utilities run on a T414 or T800 processor. This manual makes the simplifying assumption that the development environment will be an Inmos IMS B004 transputer evaluation board, or a transputer system which is largely compatible with a B004. This board is a single plug-in card for the standard IBM PC bus, with one transputer and either 1 or 2MB of RAM.

The assumption is also made here that the host computer for the B004 will be an IBM PC with a hard disk drive, or one of the many personal computers compatible with the original IBM machines.

A variety of target processors are supported by Parallel C.

- The T414 and T800 target environment is assumed to be similar to the development environment described above. Both processors are fully supported by Parallel C. However, early pre-production transputers contained faults which may cause problems with the operation of Parallel C programs. If you will

be using early transputer chips, you should check appendix B for details of the problems which you may encounter, and how to get round them.

- The T425 processor can be used with Parallel C if it is treated as if it were a T414; some additional instructions are included in this processor which are not at present accepted by the in-line assembler within Parallel C. If you wish to use these instructions in assembly-language code, you must code them using the `opr` instruction instead.
- Parallel C can also be used to build programs for the 16-bit T212 and T222 processors. Target environments for these are discussed in chapter 8.

## Document Structure

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

- *Part I: Getting Started* covers installing Parallel C on your machine and verifying that it is operating correctly.
- *Part II: Tutorial* 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.
- *Part III: Reference* contains the detailed technical information which you will require to write sophisticated applications for the transputer using Parallel C.
- 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, First Edition*[1], by Brian W. Kernighan and Dennis M. Ritchie.

As Parallel C includes so many ANSI features, it may be useful to consult the second edition[2] of this book, by the same authors, which describes the standard dialect. However, as certain ANSI features are not supported by Parallel C, beginners in particular may find the first edition preferable. Both editions are available in most bookshops or from the publishers.

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*[4]. For compatible machines made by other manufacturers, the operating system will usually be MS-DOS, described in *Microsoft MS-DOS User's Reference*[5]. These two operating systems are largely compatible, and their documentation is very similar. We will refer to “MS-DOS” in this manual to mean the operating system used on your machine. The term *DOS Reference Manual* will be used to refer to the appropriate manual.

References to these and other documents mentioned in this manual are collected in a bibliography, which can be found on page 485.

## Conventions

### Text Conventions

Throughout this manual, text printed in **this typeface** represents direct verbatim communication with the computer: for example, pieces of C text, commands to MS-DOS and responses from the computer.

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

```
t8c source-file
```

This means that the command consists of:

1. The word “t8c”, typed exactly like that.
2. A *source-file*: not the text **source-file**, but an item of the *source-file* class, for example “myprog.c”.

### Installation Directory

As we shall see in chapter 1, it is possible to install the Parallel C compiler and its associated software in any directory. By default, however, it will be installed in directory **\tc2v2**, and throughout the rest of the User Guide it will be assumed that this is what has been done. Users who have chosen to install the software in another directory should replace the directory **\tc2v2**, wherever it is mentioned, by the name of their own installation directory.





# Chapter 1

## Installing the Compiler

This chapter contains instructions on how to load Parallel C from the supplied floppy disks onto a hard disk and make it ready for use.

You can skip this chapter if the compiler has already been installed on the machine you are using.

### 1.1 Installation Directory

Before we go any further, you should decide on an *installation directory*; that is, the directory where you want the compiler and its associated software to be placed. It is best to reserve a directory just for this purpose, rather than mixing Parallel C up with other system software. In particular, do not try to install Parallel C in the same directory as any other 3L compilers, as many of the files have the same names, even though their contents may be different. The installation procedure will overwrite any file in the installation directory which has the same name as a Parallel C file.

Notice that certain files are created at installation time, and include the name of the installation directory. This means that if you wish

to move the software to another directory, you cannot simply copy it across; instead, you should install it again from the floppy disks.

The default installation directory is `\tc2v2`. In the other chapters of this *User Guide* we shall assume that this is where the software has been installed. If this is not the case, you should mentally substitute the name of your installation directory whenever `\tc2v2` is mentioned.

## 1.2 Installing the Software

The compiler is distributed on three 360KB floppy disks. The contents of these disks are described in detail in appendix A.

To install Parallel C on your hard disk, follow this procedure.

1. Place the disk labelled **Disk 1 of 3** in your floppy disk drive **A:**.
2. Type the following commands:

```
C>a:
```

```
A>install
```

3. Answer any questions the `install` program asks you. One of these will enable you to specify your installation directory (see above). If you wish to accept the default installation directory (`\tc2v2`), you should just press the Enter key in answer to this question; otherwise, erase "`\tc2v2`" and type the name of the installation directory you want.
4. Place the appropriate disks in drive **A:** when the `install` program asks for them.

It is important to use the supplied `install` program to install Parallel C. If you simply copy the files, the installation will not be performed correctly.

## 1.3 The Search Path

The compiler is now installed, but can only be run in the installation directory. Before the compiler can be used from other directories the installation directory must be added to the MS-DOS *search path*. Program files stored in directories which are on the search path can be loaded and run simply by typing the name of the program as a command. So, to make sure that the C compiler is available as a command (**t8c**, **t4c** or **t2c**), the installation directory must be added to the search path.

The search path for your machine is set up by the batch file **c:\autoexec.bat** which is automatically executed when the machine starts up. To change the path, you will need to edit the **autoexec.bat** file using a text editor like **edlin**. (The *DOS Reference Manual* explains how to use **edlin**). Your **autoexec.bat** file will probably already contain a line of the following form:

```
path ... list of directories ...
```

For example:

```
path c:\dos;c:\utils
```

In this case, you will need to add the text “**c:\tc2v2**” (if that is the installation directory) on to the end of the line, giving:

```
path c:\dos;c:\utils;c:\tc2v2
```

If there is no **path** line in the **autoexec.bat** file, just add the line:

```
path c:\tc2v2
```

Some important points about setting the search path should be noted:

1. The documentation for previous versions of 3L compilers, including Parallel C, recommended the use of a **set path=** command to set up the search path. This is equivalent to the **path**

command, and can be changed to include your installation directory in the same way.

2. If you already have earlier C compiler installed on your machine, its installation directory may appear in your search path. It should be removed from the search path before adding the installation directory of the new version.
3. If you are a user of the Inmos TDS environment, your search path will probably include a reference to the directory where the TDS is held, such as `\tds2dir`. This reference must not precede the Parallel C installation directory in the path; if it does, the wrong version of the **afserver** program will be called.
4. From time to time, 3L release new versions of components, such as the linker or the **afserver**, which are included in more than one compiler product. This means that if you are a user of any other 3L compilers, you should make sure that the installation directory of the latest compiler product precedes all the others. This will ensure that the latest versions of these common components are picked up; they will be compatible with all the compiler products.

Once your **autoexec.bat** file has been changed, you will need to reboot your machine to make the changes effective.

## 1.4 Environmental variable 3LCC\_INC

Sometimes it is also necessary to define the MS-DOS environmental variable **3LCC\_INC** when you install the compiler. **3LCC\_INC** can be used to define where the compiler should look for the header files which are included in programs by lines such as

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

If `3LCC_INC` is not defined, the compiler looks for header files, such as `stdio.h`, in directory `\tc2v2` on the current disk. This means that `3LCC_INC` must be defined in the following circumstances.

- If you have decided to install the compiler in a directory other than `\tc2v2`.
- If you use the compiler from a disk other than the one where the compiler is installed.

In either case, you should include in your `autoexec.bat` file a line of this format:

```
set 3LCC_INC=f:\lib\ThreeLC
```

A full discussion of how the `#include` directive is handled may be found in section 9.6 in part III of this manual.



## Chapter 2

# Confidence Testing

This chapter describes a short procedure which may be followed to check that installation has been done correctly.

1. Set the current disk drive to the one on which Parallel C has been installed. For example, if the compiler has been installed in directory `c:\tc2v2`, do this:

```
D>c:
```

```
C>
```

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

```
C>cd \mine
```

```
C>
```

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

3. Check that the correct versions of the **afserver** program and of the compiler are available, by typing the following command. You should see the output shown.

```
C>t8c /i -:i
IBM PC Filer server Inmos V1.3 (14th October 1987) / 3L
V1.3.7
Copyright INMOS Limited, 1985
Transputer C compiler, CC_transputer V2.2.2
Copyright (C) 3L 1991

C>
```

If the above message does not appear, check the installation procedure, and in particular, ensure that the correct **path** command has been set up.

If, after the **afserver**'s identity, the computer outputs the following, or something similar—

```
Last command = 0
Server terminated: bad protocol when expecting INT32
```

—it is likely that there has been some error in setting up the transputer board. In particular, please check that the wire links, accessible from the back of the PC, have been correctly installed. The transputer board's documentation should help with this.

4. Copy the example **hello.c** file to the current directory. If the installation directory is **\tc2v2**, for example, you should type this:

```
C>copy \tc2v2\examples\hello.c
1 File(s) copied

C>
```

5. Compile the example using the T8 version of the compiler (this will work for the T4 as well, because the example contains no



floating-point instructions):

```
C>t8c hello
```

```
C>
```

6. Link the resulting binary file with the necessary parts of the run-time library, and the harness:

```
C>t8clink hello
```

```
C>linkt      hello \tc2v2\crt1t8 \tc2v2\t8harn
```

```
C>
```

7. Finally, the program can be run:

```
C>afserver -:b hello.b4
```

```
hello, world
```

```
C>
```

The output “**hello, world**” comes from the `hello.c` example program. If it does not appear, we recommend that the installation procedure should be carefully repeated, and the confidence test procedure followed again. If this message still does not appear, please contact your dealer for further assistance.



## Chapter 3

# Developing Sequential Programs

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

The instructions in this chapter assume that the C compiler has already been installed as described in chapter 1.

The operating procedures for the T2 differ in some ways from the ones discussed here, which are appropriate for the T4 and T8 transputers. If you are developing programs for the T2, you should read this chapter for general information, and then study chapter 8.

Some of the procedures described here are different for T4 and T8 transputers. You should find out which type of transputer is fitted in your PC before using the compiler.

## 3.1 Editing

Any editor which handles standard MS-DOS text files can be used to create or change C source programs. The example below shows how the `edlin` editor supplied with MS-DOS can be used to create a new C source program.

```
C>edlin hello.c
New file
*i
    1:*main()
    2:*{
    3:*    printf("hello, world\n");
    4:*}
    5:*^C
*e
C>
```

The *DOS Reference Manual* explains how to use `edlin`.

Note that the “folded” files which the Inmos TDS works with are not ordinary MS-DOS text files and that therefore they cannot be used directly as input to the compiler. However, the `tdslist` utility program supplied with the TDS will convert TDS-format text files into ordinary MS-DOS text files which can be read by Parallel C.

## 3.2 Compiling

A C source program is compiled into a binary object (`.bin`) file of T8 transputer instructions by a command of the form:

```
t8c source-file
```

To compile code for a T4 transputer, use the command

```
t4c source-file
```

Note that, in general, code compiled for a T4 will not run on a T8 (or vice versa) so you must use the command appropriate for the type of processor in your transputer board.

The *source-file* is the filename of the C source program which is to be compiled. If no filename extension is given in the command, `.c` is added automatically.

So, to compile the file `hello.c` for the T8, you would give the command:

```
C>t8c hello
```

If the source file contains no errors, an output object file `hello.bin` is produced. If the compiler detects errors in the source program, it writes diagnostic messages to the MS-DOS standard output stream. Error messages may therefore be redirected using `>`, or piped using `|`, as described in the *DOS Reference Manual*. The format of compiler error messages, and a list of all the messages which may be produced by the compiler, appears in section 9.9 in part III of this manual.

### 3.3 Linking

Once a Parallel C program has been compiled into an object (`.bin`) file, it must be linked with any external functions it requires before it can be run, including functions like `printf` and other functions from the Parallel C run-time library. This is done by the *linker*. Here we discuss the most usual linker operations; a full description of the linker can be found in chapter 12.

Rather than calling the linker directly, it is usually more convenient to use one of the batch files provided for the purpose.

To link T4 code produced by the `t4c` compiler use the command:

```
t4clink object-file
```

For example,

```
t4clink hello
```

To link T8 code produced by **t8c** use the command:

```
t8clink object-file
```

You must use the link command appropriate to the target processor (T4 or T8).

Both batch files automatically append **.bin** to the object file name and produce an executable file with the same file name as the object file and extension **.b4**.

### 3.3.1 Linking More than One Object File

This section deals with linking more than one object file at a time. If you only want to link single object files for now, you can skip to section 3.4 which describes how to run executable files produced by the linker.

The **t4clink** and **t8clink** batch files can be used to link up to nine object files. As before, the extensions of all the object files are assumed to be **.bin**. The executable file generated will have the file name of the first object file specified, with the extension **.b4**.

For example, if there are two C source files, **main.c** and **fns.c**, the following commands will compile them and link them together, producing an executable file for the T4 called **main.b4**.

```
C>t4c main
```

```
C>t4c fns
```

```
C>t4clink main fns
```

Compiling and linking the example files above for the T8 would be done as follows:

```
C>t8c main
```

```
C>t8c fns  
  
C>t8clink main fns
```

### 3.3.2 Indirect Files

It is quite common for programs to consist of many different object files. The `t4clink` and `t8clink` batch files cannot handle more than nine, but even with fewer files than this, you may find the command line awkward to type.

The linker provides a way of getting round this problem, called an *indirect file*. An indirect file is a text file containing a list of object file names, all of which are to be included in the executable file. It is specified in the linker command by its file name preceded by an '@'. For example:

```
C>t8clink @objfiles
```

This will cause the linker to find the file `objfiles.dat`, and link together all the object files specified in it. As usual, the generated file will be given the name of the first object file with the extension `.b4`.

Indirect files are assumed to have the extension `.dat`. They contain a list of MS-DOS file names, with one file name on each line. Full path names, including directory specifications, are allowed. Indirect files may also include the names of other indirect files, by preceding with an '@'; nesting indirect files in this way may be done to five levels.

The example indirect file `objfiles.dat` above might contain the following text:

```
main  
fns  
\userlib\general\io  
@grafpack
```

When used in the example given above, this will link the object files `main.bin` and `fns.bin` from the current directory and `io.bin` from the directory `\userlib\general`, together with all the object files specified in the indirect file `grafpack.dat`. The executable file generated will be `main.b4`.

### 3.3.3 Calling the Linker Directly

Occasionally, instead of using the batch files, you may need to call the linker directly, or write your own batch files to do so. Fuller information about the linker may be found in chapter 12, and details of the internal format of object files are provided in the Inmos *Stand-Alone Compiler Implementation Manual*[14].

The linker is invoked by the command `linkt`. The general form of a link command is

```
linkt object-files , executable-file
```

*object-files* is a list of object file names separated by spaces. These are the object files which are to be linked together. All of them must have been compiled for the same processor type (T4 or T8). If an object file is specified without an extension, the extension is assumed to be `.bin`.

The order in which the object files are specified is significant. Details of this may be found in sections 3.5 and 9.4.4.2.

The *executable-file* is the name of the file to which the linker writes the executable output code. If no extension is specified, the linker supplies the extension `.b4`. The executable file and its preceding comma may be omitted; in this case, the executable file is given the same file name as the first object file in the command line, with the extension `.b4`. If the first file mentioned on the command line is an indirect file, the executable file is given a name taken from the name of the first object file listed in the indirect file.



To link C programs, you must include in the list of object files both the Parallel C run-time library and a special object file called a “harness”. The directory `\tc2v2` contains two versions of both of these components: `crt1t4.bin` and `t4harn.bin` for T4 transputers, and `crt1t8.bin` and `t8harn.bin` for T8 transputers. The linker will not allow you to mix T4 and T8 object files.

The example below shows the command necessary to link all the files listed in the indirect file `subs.dat` into a single executable file for the T4, called `prog.b4`.

```
C>linkt @subs \tc2v2\crt1t4 \tc2v2\t4harn,prog
```

Note that the Parallel C run-time library (`crt1t4.bin`) and the harness (`t4harn.bin`) must both be named explicitly as input object files.

For the T8, the command would be the following.

```
C>linkt @subs \tc2v2\crt1t8 \tc2v2\t8harn,prog
```

### 3.3.4 Libraries

It is often convenient to be able to treat a group of object files as a single unit. For example, the Parallel C run-time library consists of many separate object files, but is supplied as a single file containing all of them.

The linker provides the option of linking together a group of object files to produce a *library* file instead of an executable file. The library contains all of the code and entry points defined by the input object files, which can be changed or deleted without affecting the library. To change a library it must be relinked from its component parts.

Library files have several advantages over using indirect files.

- The linker selects from the library file only those modules which are actually referenced elsewhere in the program; the others are not included in the executable file.

- Copying a single file to another place is simpler than copying many component object files and making sure that the corresponding indirect file is kept up to date with changes in directory and file names.
- Opening just one library file is faster than opening an indirect file and several object files.

However, using an indirect file may be faster while a library is being developed because there is no need to relink the library whenever a component module is changed.

A linker command of the form shown below is used to produce a library from a number of component object files.

```
linkt object-files, library-file/l
```

The option letter after the ‘/’ is a lower case ‘L’.

The form of the input *object-files* is the same as for normal operation of the linker: a list of filenames separated by spaces. Indirect files are indicated by an ‘@’ sign as before.

The *library-file* must be a single MS-DOS file name. If no extension is specified, the linker will give it the extension `.lib`. Note that this is different from the default extension which the linker uses for libraries when they are specified as input files, which is `.bin`.

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 linker commands.

```
C>type graflib.dat
core
tek
hp

C>linkt @graflib,graflib.bin/l

C>type myprog.dat
```

```
myprog
graflib
\tc2v2\crt1t8
\tc2v2\t8harn

C>linkt @myprog
```

### 3.4 Running

Executable programs are loaded into the transputer board and run using the **afserver** program, which runs on the IBM PC.

The **afserver** is an ordinary MS-DOS program, and after loading the C program into the transputer board, it remains active throughout the program's run. Instructions are sent from the C run-time library to the **afserver** whenever it needs to perform MS-DOS functions such as reading information from the disks, displaying output on the screen and so on. The results of these operations are sent by the **afserver** back to the transputer board.

The command to load and run a program is:

```
afserver -:b filename
```

The *filename* must be the name of an executable file produced by the linker. The file name extension must be specified. An example of a command to load and run a simple program would be:

```
C>afserver -:b hello.b4
```

Note that this will only work if your program uses a fairly small amount of stack memory. See section 3.5 for information about running programs with larger stack requirements.

Appendix section D.3 includes more information about the **afserver** and its options, and the Inmos *Stand-Alone Compiler Implementation Manual*[14] (section 10) contains a full description. Note that the **-:e** (test error flag) switch described in [14] is not supported

for use with Parallel C programs. For improved performance, the C compiler relies on being able to generate code which might incidentally cause the error flag to be set. Therefore, the transputer error flag may be set as part of the normal execution of a C program.

The running of programs can be simplified by putting the appropriate **afserver** command into an MS-DOS batch file. Typing the name of the batch file is then sufficient to run the program. For example:

```
C>type myprog.bat
afserver -:b \mydir\myprog.b4

C>myprog
```

The command **myprog** will then call **afserver** to load the executable file **\mydir\myprog.b4** into the transputer board and start it. Note that if a program compiled and linked for the T4 is loaded into a T8 (or vice versa) the effects will be unpredictable.

### 3.4.1 Using C Programs as MS-DOS Commands

Because of the limitations on what can be done with MS-DOS batch files it is useful to have a way of running a transputer C program as if it were an MS-DOS **.exe** file.

You can turn any **.b4** file into an MS-DOS command by making a copy of the file **\tc2v1\linkt.exe** in the same directory as the **.b4** file, giving it the same root filename as the **.b4** file but keeping the **.exe** extension. For example, if the current directory contains the executable file **ex.b4**, it can be run as a command by typing:

```
C>copy \tc2v1\linkt.exe ex.exe

C>ex
```

This new **ex** command can be used from any directory, provided the directory containing **ex.exe** and **ex.b4** is on the MS-DOS search path.

(**linkt.exe** works by taking the command verb from its command line, adding **.b4**, and then calling **afserver** to load that file from the same directory **linkt.exe** itself was loaded from).

When a **.b4** file is invoked via a “driver” program in this way, the **-:o 1** option (see section 3.4) is added automatically and the program is given a large amount of stack space. If you want to run a program as an MS-DOS command, but with its stack in fast on-chip RAM, you should invoke the program as usual but add **-:o 0** to the command line (hyphen, colon, letter ‘o’, then a space followed by the digit zero). For example:

```
C>ex -:o 0
```

### 3.4.2 Command-Line Arguments

The **afserver** passes its command line on to the user program it invokes, for use as program arguments. For example:

```
C>afserver -:b myprog.b4 fred
```

Here, the character string “fred” is passed on to **myprog.b4**.

Note that the “**-:b myprog.b4**” part of the command is not passed through as an argument to **myprog.b4**. In general, **afserver** option switches (**-:b**, **-:o**) and their arguments are not passed on to the user program. Any DOS file redirections (see section 3.4.3 below) are also stripped out.

The text of the command line is also passed on to the user program if the **afserver** is invoked using the driver program described in section 3.4.1. For example:

```
C>myprog xyz abc
```

Here, the program argument string “xyz abc” is passed on to **myprog.b4**.

The program argument string is broken up into a sequence of tokens before being passed to the C **main** program function. Tokens

are separated by blank or horizontal tab characters, so in the first example there was one token: “**fred**”, and in the second example there were two: “**xyz**” and “**abc**”.

When the C `main` program function is called, it is passed the following arguments:

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

`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.

### 3.4.3 I/O Redirection and Piping

Normally the C standard input stream (`stdin`) read by functions like `getc` and `scanf` is the keyboard. Standard input can be taken from a file by using the MS-DOS redirection symbol ‘<’ in the normal way. For example, to use the file `chap1.txt` as the standard input stream for a word counting program `wc.b4` you could use the command:

```
C>afserver -:b wc.b4 <chap1.txt
```

This also works if `wc.b4` is invoked by a driver program, `wc.exe`:

```
C>wc <chap1.txt
```

Similarly, the standard output stream (`stdout`) written by functions like `putc` and `printf` is normally the screen. Standard output is redirected using the ‘>’ symbol. A program called `cat.b4` which concatenated the contents of all the input filenames given as its program arguments and wrote the result to the standard output

stream could be used to concatenate the files `a.txt`, `b.txt` and `c.txt`, writing the result to another file `all3.txt` as follows:

```
C>afserver -:b cat.b4 a.txt b.txt c.txt >all3.txt
```

Note that neither “>*filename*” nor “<*filename*” is considered to be part of the program arguments; these special forms do not appear in the `argv` array passed to a C main program.

Standard output may also be *pip*ed into an MS-DOS *filter* program by writing the name of the filter after a vertical bar ‘|’, as shown below.

```
C>afserver -:b cat.b4 a.txt b.txt | more
```

The *DOS Reference Manual* describes in detail what can be done with filters. (The `more` program simply displays its input on the screen, a page at a time).

## 3.5 Memory Use

The memory used by a C program is divided into four storage areas.

- *Code storage* is used to hold the executable instructions of the program itself, together with some constant data and control information.
- *Static storage* is used to hold static and external variables, including variables declared at the global level.
- *Stack storage* (sometimes referred to as *workspace*) is used for `auto` variables. The stack is also used for function calls and passing parameters.

In addition, library functions use varying amounts of stack space as working storage. The stack requirements of the mathematical functions are given in the Inmos *TDS Compiler Implementation Manual*[15] (Section 10, Parameters and workspace

requirements) and are generally about 40 to 100 words. The stack requirements of the floating-point arithmetic support library for the T4 are generally about 10 to 40 words. About 70 words of stack storage are permanently reserved for use by the run-time library.

- *Heap storage* is used to hold all variables created by calls on `malloc`, etc. It is also used internally by the run-time library for I/O buffers, etc.

These four areas of storage are mapped onto two areas of physical memory:

- *On-chip* memory. The T4 has 2KB of fast on-chip memory, and the T8 has 4KB.
- *External* memory. The Inmos B004 board has either 1MB or 2MB of external memory.

Using the linker only, two methods of mapping the storage areas onto physical memory are available: the default method, and the alternative method. You can select the method you wish to use by calling the `afserver` in different ways, which are discussed below.

The configurers required for developing parallel programs give the user more advanced methods for controlling the use of memory. See section 5.8, and chapter 18.

### 3.5.1 Default Memory Mapping

Default memory mapping is used if the `afserver` program is called as described in section 3.4 above. With this arrangement, the T4's on-chip memory, and the first 2KB of the T8's on-chip memory, are used for stack storage. Since on-chip memory is faster than external memory, programs can run much faster with default memory mapping. Obviously, you must be certain that the program's stack storage will fit in the available 2KB.



If you are using a T8, default memory mapping provides an opportunity for further speed improvements, since the remaining 2KB of the T8's on-chip memory is available for code storage. To take advantage of this, you should place small, speed-critical subprograms at the beginning of the link-list.

*WARNING: A program which exceeds the amount of available stack space will fail in unpredictable ways: it may hang, or it may simply give wrong answers.*

### 3.5.2 Alternative Memory Mapping

Unless you are sure your program's stack data will fit into the 2KB of available on-chip memory, you should use the alternative method of memory mapping. This is done by calling the **afserver** like this:

```
C>afserver -:b myprog.b4 -:o 1
```

With the alternative method, the stack is placed in external memory, and so is limited only by the amount of external memory available. On the T4, on-chip RAM is not used at all. On the T8, although the upper 2K of on-chip RAM is used for code as before, the rest of it is unused.

The program will execute more slowly with this method, because external memory is slower than on-chip memory.

Note that the **afserver** switch is typed as hyphen, colon, option letter 'o', then a space, then the digit one.

### 3.5.3 Limit on Program Memory

The current version of the linker generates executable files which will only run correctly on boards having 1MB or 2MB of memory. To get round this restriction, the Parallel C kit includes the **mempatch** program which may be used to change executable files to run on

boards which have different amounts of memory. See chapter 13 for a discussion of `mempatch`.

## Chapter 4

# Introduction to Parallel C

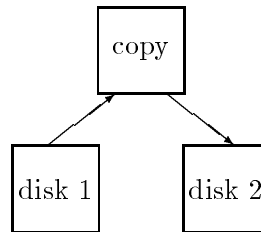
This chapter aims to help you become familiar with Parallel C and its terminology. If you know occam, or if you have read a lot about the transputer, then you will already be familiar with the ideas on which Parallel C is based. If not, don't worry; the ideas are quite simple. They are explained in outline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

### 4.1 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 a bit of hardware or a software module; in particular it may also be another complex system, itself consisting of a number of communicating processes.

## 4.2 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<sub>16</sub>. 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.

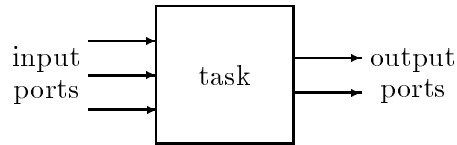


Figure 4.1: a task viewed as a “black box”.

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.

### 4.3 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. The code of a task is a single transputer image (`.b4`) file generated by the ordinary linker, `linkt`.

Tasks can be treated as software “black boxes” connected together via their ports, as shown in figure 4.1.

```

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

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

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

```

Figure 4.2: Complete example task with one input and one output port.

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 4.2.

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 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 multiplexer tasks.

## 4.4 Simultaneous Input

All of the code of a task can be written in an ordinary sequential language like C, except for one extra feature needed by languages based on the communicating sequential processes idea. This extra feature is a way of making a process wait until a message is received on any one of a number of input channels. For example, the main loop of a file server process would want to wait until a message was available from any one of its "client" processes. It can't read them all sequentially because a message could come from any one of them, in any order.

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



## 4.5 Parallel Execution Threads

Parallel C supports *multi-threaded* tasks. Tasks dynamically create new execution threads by passing a pointer to a function and an amount of stack space to a library function. The new execution thread then starts executing the code of the pointed-to function concurrently with the thread which created it. The new thread runs in the same context as its creator; they share their **static**, **extern** and heap memory areas. The only private storage available to the new thread is its stack. The parent thread has no direct control over its offspring, which continues to execute until it terminates itself by returning from the function which was invoked, or by calling another library function.

Parallel C's threads resemble the execution threads of OS/2, the “processes” of Modula-2, and the “coroutines” of some other languages. Each one has its own stack but shares the rest of its data with all the other threads in the same task.

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

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

## 4.6 Configuring an Application

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

transputers?

First, each individual task is built by compiling all its source files with the C compiler and using the linker (`linkt`) to combine the resulting binary (`.bin`) files with the Parallel C run-time library to produce a task image (`.b4`) file.

Now a bootable application image file must be generated from the component task (`.b4`) files. The program which does this is called the *configurer*. It is driven by a user-supplied *configuration file* which specifies:

- the hardware configuration (processors, and the wires connecting them) on which the application is to be run;
- the names of the `.b4` files containing the component tasks of the application;
- the connections between the various tasks' ports;
- the placement of particular tasks onto particular processors in the physical network.

The output of the configurer is an application file which can be booted into the specified hardware network and run using the same `afserver` program used for simple stand-alone programs.

## 4.7 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; repeat.

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 **afserver** 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.

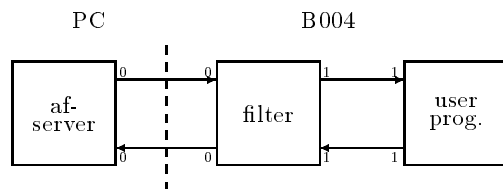


## Chapter 5

# Developing Parallel Programs

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 a PC runs in parallel with the `afserver` program on the host computer, as shown below.



The **afserver** task is an ordinary MS-DOS executable (**.exe**) file that runs on the PC. It loads executable **.b4** files into the transputer and also acts as a file server, handling I/O requests made by the transputer. The **afserver** and the transputer execute in parallel and communicate via an Inmos link. The messages sent to the **afserver** are normally generated by the Parallel C run-time library. It converts I/O operations like **putchar** and **fprintf** into messages requesting the **afserver** to perform MS-DOS operations like *write 512 bytes* and then waits for the **afserver** to reply.

In principle, the **afserver** task could be directly connected to the user program. In practice, a *filter* task is interposed between them. The filter runs in parallel with the **afserver** and the user task; it simply passes on messages travelling in both directions. The filter is required because sometimes the messages passed between the user program and the **afserver** are only one byte long and the revision A T414 chip cannot handle single-byte message transfers on its hardware links. The filter pads out 1-byte messages to 2 bytes to avoid this problem.

## 5.1 Configuring One User Task

Up to now a standard “harness”, **t4harn.bin**, 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 (**afserver**, filter and one user task) to the harness; the configuration is assumed.

Using the standard harness is simple but inflexible. We need a way to produce executable files for more complicated system configurations containing many tasks and many transputers. The configurator program supplied with the compiler can do this; a simpler harness (known as the “task harness”) can then be used.

The configurator is driven by a user-written *configuration file* which describes the system to be built: the file lists all the physical proces-

```

!
! UPPER.CFG
!
processor host      !the PC
processor root      !the transputer in the B004
wire  jumper -     !connects...
      root[0] -     !link 0 of root transputer
      host[0]      !to the PC bus

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

place afserver host !afserver runs on PC
place upper root    !everything else on transputer
place filter root

connect ?  filter[0]  afserver[0]
connect ?  afserver[0] filter[0]
connect ?  filter[1]  upper[1]
connect ?  upper[1]   filter[1]

```

Figure 5.1: Configuration File with One Example Task

sors 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 5.1. In the rest of this section we will look at its contents in detail.

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 5.2 (the corresponding configuration file is called `upper.cfg`). Note that the examples discussed here are not the same as the files with the same names supplied in the distribution kit.

```

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

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

```

Figure 5.2: C Source File for Upper Casing Program, `upper.c`

### 5.1.1 Hardware Configuration

The first thing the configuration needs to describe is the hardware configuration. A single B004 board plugged into a PC is very easy to describe.

```

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

```

There are two processors: the host PC and the root transputer in the B004. The root transputer is so called because if a larger network is built around a basic B004 system, the transputer directly connected to the PC becomes the root of the network—all communication with the file system on the PC must pass through it.

A wire connects the root transputer's link 0 to the host processor. The WIRE statement describes actual physical cables, in this case the little jumper you have to plug into the back of a B004 board which connects link 0 on the transputer to the PC bus. 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 above with the full configuration file in figure 5.1. 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.

### 5.1.2 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.

#### 5.1.2.1 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 **afserver** has only one input port (for file system requests) and one output port for responses.

```
task afserver 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 **afserver** 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. For ready-made tasks supplied with the compiler, like **filter**, memory requirements can be looked up in the data sheets in chapter 20.

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 5.8 below gives hints on estimating memory requirements in cases where multiple user-written tasks must be placed on the same processor.

### 5.1.2.2 Assigning Tasks to Processors

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

```
place afserver host
```

```
place upper root
place filter root
```

### 5.1.2.3 Making Connections between Tasks

The CONNECT statement establishes a channel between two tasks, by connecting an output port to an input port. Because channels (unlike wires) are unidirectional, two CONNECT statements are needed to create channels going in both directions between the **afserver** and the filter.

```
connect ? filter[0] afserver[0]
connect ? afserver[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.

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

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

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

### 5.1.3 Building the Application

Once a configuration file has been written all we have to do to execute the application is compile the C source file **upper.c**, link the resulting

object file with the C run-time library, and then run the configurer.

The example below shows what must be done to build an executable file from the uppercasing example:

```
C>t4c upper

C>t4ctask upper

C>config upper.cfg upper.app
WARNING: no memory allocation specified for task upper:
        assuming rest of processor's memory

C>afserver -:b upper.app
case changer
CASE CHANGER
^Z

C>
```

Two commands are new: `t4ctask` and `config`.

### 5.1.3.1 Linking for the Configurer

The ordinary batch file for linking C programs (`t4clink`) is not suitable for linking a task because it links in the standard harness. `t4ctask.bat` is a batch file supplied with the compiler which links an object (`.bin`) file with the Parallel C run-time library and a vestigial *task harness* containing neither the filter process nor any system initialisation code. The example below shows two C source files, `main.c` and `subs.c`, being compiled and then linked together to form a T4 task called `main.b4`.

```
C>t4c main

C>t4c subs

C>t4ctask main subs
```

Like `t4clink`, the `t4ctask` batch file can handle up to nine object files on the command line. If you need to link more files than this,

you will need to use an indirect file, as described in section 3.3.2. If you need to call the linker directly, as described in section 3.3.3, you must link in the run-time library, `crtlt4.bin` and the task harness, `taskharn.t4`, by hand. Both can be found in the installation directory, `\tc2v2`.

As usual, there are T8 versions of the batch file and the task harness. They are called `t8ctask` and `taskharn.t8`.

*Note: it is important to link all tasks which are to be used with the configurer with the correct harness. If the wrong harness is used (for example by accidentally using `t4clink` rather than `t4ctask`) then the configured application will fail to operate correctly. It may fail to execute, or it may simply give wrong answers.*

### 5.1.3.2 Running the Configurer

The configurer is invoked by the `config` command. Two filenames must be specified on the command line: first the configuration file, then the name of the executable file to be output. For our case-conversion example, the required `config` command line was:

```
C>config upper.cfg upper.app
```

The configurer does not supply default filename extensions, but `.cfg` is conventional for configuration files.

File names for the task images which make up the application are not supplied on the command line; the configurer derives them automatically by appending `.b4` to the task identifiers given in the configuration file. In our example, the configurer will search for task image files called `upper.b4` and `filter.b4`.

If a task image file is not found in the current directory, the configurer will automatically search all of the directories on the MS-DOS search path, so there is no need to make copies of ready-made tasks like `filter.b4` held in the same directory as the compiler (`\tc2v2`).

The search path can be modified in the usual way by the MS-DOS commands **path** and **set**.

This automatic mechanism for specifying task image file names can be overridden by the **FILE** attribute of the configuration language's **TASK** statement, described in chapter 18.

Note that tasks placed on the **host** (PC) processor are not searched for in this way to be included in the output application file. The configurer does not attempt to load **afserver.b4** into the PC from the transputer! The **afserver** task must be declared and placed on the **host** simply in order to give a name to the object with which the **filter** task communicates over its port 0. However, **afserver.exe** will always be running in the PC, ready to accept file I/O requests, when a transputer application starts running, simply because the **afserver** is used to load the application into the transputer. It is therefore reasonable to regard it as part of the configuration.

The output from the configurer can be run directly using the **afserver**:

```
C>afserver -:b upper.app
```

The actual hardware configuration of the transputer network attached to your PC must match the declarations in the configuration file. The memory requirements of configured tasks are specified in the configuration file; the **afserver** options  **-:o 1** and  **-:o 0** are ignored by configured applications.

## 5.2 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.

### 5.2.1 Inter-Task Communication Functions

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

C tasks send messages using the channel I/O functions described in chapter 10. The `chan` package provides functions to send and receive messages of any length. The driver task is shown in figure 5.3; it uses `chan_in_word` and `chan_out_word` to handle word-sized messages. A word is the same size as an `int`.

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, `\tc2v2`.

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.

```

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

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

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

```

Figure 5.3: `driver.c` with Channel I/O Calls

Beware when using the channel I/O functions that sending and receiving tasks must always agree on the size of messages. For example, if a task sends a 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 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”.

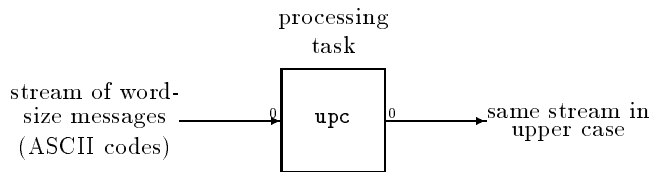
The number of output ports in the vector is defined by the OUTS attribute of the TASK statement used to declare the task in the configuration file. Our **driver** task has **outs=3**, so there are three elements in its output port vector, numbered 0 to 2.

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 multiplexer 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 5.4.

The processing task uses the same channel I/O functions as the driver to send and receive messages. It terminates when it receives a  $-1$

```

/*
** upc.c stand-alone processing task;
** communicates with driver.c
*/

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

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

    for (;;) {
        chan_in_word(&c, in_ports[0]);
        if (c == EOF) break;    /* terminate task */
        chan_out_word( toupper(c), out_ports[0] );
    }
}

```

Figure 5.4: The Processing Task

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 (see figure 5.1) 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 5.8 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.

## 5.3 Building Multi-Task Systems

We will run into a problem when trying to compile and link the components of the dual-task system.

The ordinary C run-time library expects to send messages to the **afserver** on output port 1 and receive replies on input port 1. This is true even if your C program does not explicitly use any standard I/O functions like **printf**—the library will still try to open the standard input and output streams, and read the command-line string from the host machine in order to initialize **argc** and **argv**.

This means that even though it does no C I/O, the **upc** task will still attempt to communicate with the **afserver** if it is linked with the standard run-time library. However, the **afserver** is already connected to the driver task. The **afserver** task can't simply be shared between the driver and **upc** tasks, because that would require connecting one port on the **afserver** task to two client ports. That is not allowed—channels must always connect one port to exactly one other port<sup>1</sup>.

This is not as restrictive as it seems, because a *stand-alone* version of the C run-time library which does not need to communicate with the **afserver** is supplied with the compiler. The stand-alone library is just the same as the ordinary library except that all the functions which require **afserver** support (I/O, date, DOS calls, etc.) have been omitted.

A multi-task application is normally split up into an I/O task with **afserver** support and one or more processing tasks which do not need ordinary C I/O because they use the channel I/O functions like **chan\_in\_word** to communicate with the I/O task.

Our example application is already in the right form: all we need to do is link the driver task with the standard run-time library and link the processing task, **upc**, with the stand-alone library.

---

<sup>1</sup>There is, in fact, a method to allow tasks to share the **afserver**. It is described in chapter 6.

In practice this logical organisation of an I/O task serving a number of parallel computing tasks is commonplace anyway.

For embedded systems which do not need disk I/O support, it is possible to link all of the component tasks with the stand-alone library, producing a consequent reduction in code size due to the absence of I/O initialisation code from the stand-alone library.

If an application includes several tasks which need to write to the screen using `printf`, or access disk files using `fread`, you will need the *global I/O* facility described in chapter 6. Normally it is just as simple to cast an application in the form of an I/O task serving multiple “compute” tasks which only use the stand-alone library and communicate by passing messages.

A batch file analogous to `t4ctask` is provided for linking an object file with the stand-alone library. It is called `t4cstask.bat`; a T8 version (`t8cstask`) is also supplied. As usual, these batch files can be used to link up to nine object files; if you need to drive the linker yourself, the files to link with are `sacrt1t4.bin` and `taskharn.t4` in the installation directory, `\tc2v2`, or their T8 equivalents. The commands required to link and configure the upper case example for a T4 are shown below.

```
C>t4c driver
```

```
C>t4ctask driver
```

```
C>t4c upc
```

```
C>t4cstask upc
```

```
C>config upc.cfg upc.app
```

```
WARNING: no memory allocation specified for task driver:
         assuming rest of processor's memory
```

```
C>afserver -:b upc.app
```

```
xyz123
```

```
XYZ123
```

```
pqr
```

```
PQR
~Z
```

You can try this out for yourself by making a copy of the relevant files, which are supplied in the directory `\tc2v2\examples`, together with the batch files `upct4.bat` and `upct8.bat`, for building the application.

## 5.4 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 **afserver**.

When interpreting a **CONNECT** statement, the configurer makes a direct channel connection between the ports, if the two tasks are in the same processor. Now they are on different processors, the channel will use the external links, and will be mapped by the configurer onto the external connection as specified in the **WIRE** statement.

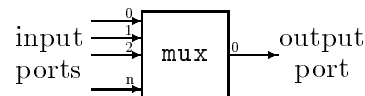
```
C>config upc.cfg upc.app
WARNING: no memory allocation specified for task driver:
        assuming rest of processor's memory

C>afserver -:b upc.app
two transputers...
TWO TRANSPUTERS...
~Z
```

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.

## 5.5 Simultaneous Input

One thing we have not yet seen how to do is to wait for a message from any one of a number of concurrently executing tasks. For example, a multiplexer 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.

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.

The `alt` functions supplied with Parallel C provide this facility. The `alt` package is more fully described in chapter 10; the interfaces to the individual functions are described in the alphabetical list of functions in chapter 11.

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

`alt_wait_vec` waits for a message to arrive on any of the channels pointed to by the array, in this case any of the multiplexer task’s input ports. It then returns the index in the array of the channel on which the message was received. If more than one message arrives at the same time, the system will choose which one to handle first. If no message ever arrives, the function will never return.

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

Figure 5.5 shows the code of the multiplexer task.

```

/* altmux.c:  message multiplexer using 'alt' package */

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

main(int argc, char *argv[], char *envp[],
     CHAN *in_ports[], int ins, CHAN *out_ports[], int outs)
{
    char buf[1024]; /* message buffer */
    int i;          /* input port on which message received */
    int msglen;     /* number of bytes in message */

    for (;;) {      /* read messages forever */

        /* wait till next message received on any input port */
        i = alt_wait_vec(ins, in_ports);

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

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

```

Figure 5.5: Multiplexer Task Using `alt` Package



## 5.6 Multi-Threaded Tasks

### 5.6.1 Creating Threads

The `alt` functions allow a limited amount of parallelism or non-determinism to be introduced into a sequential task. Parallel C also allows tasks to be fully multi-threaded. This means that a task can contain any number of concurrent processes each of which is independently executing the code of the task. All the threads in a task share the same `static`, `extern` and heap data. The threads can still operate independently because each one is given its own stack for `auto` variables. New threads are created dynamically by calling the library function `thread_create`. All of the library functions discussed in this section are described more fully in chapters 10 and 11.

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

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

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

Figure 5.6 shows a task which creates just two threads, a `producer` thread which generates a sequence of word-sized messages and a `consumer` thread which processes them. The messages are transmitted across an internal channel, `chan`. The channel transmits the data,

and also ensures synchronization: the `consumer` cannot proceed once it has called `chan_in_word` until the `producer` sends a message over `chan`. Similarly, if the `consumer` thread is busy when the `producer` attempts to send a message, it will be blocked until the `consumer` comes to read its next message.

There are several points to note about this example.

First, any channels to be used to synchronize the operations of multiple threads should be declared and initialized before those threads are created.

Second, it is a bad idea for the `main` function of a task to return while any threads it has created are still active if, as in this case, one of the threads may use C standard I/O. If this happens, the `main` function may exit, causing the run-time library to shut down the I/O system and close all open files before some thread which needs to do I/O has finished. To forestall this possibility, an extra channel has been added in the example from the `consumer` thread back to the original `main` thread. It is used purely for synchronization. When the consumer thread is about to terminate, it sends a dummy message over this channel. The `main` thread waits for this message before returning.

Finally, note the use of `par_printf` in place of `printf` in the `consumer` thread. If multiple threads are active in a task, and more than one thread may need to call the run-time library, then their calls must be interlocked using a semaphore. The `par` package provides ready-interlocked versions of some common functions like `printf`. The interlock is not actually necessary in this case, since no other thread will be attempting to use the run-time library at the same time, but it is as well to be aware of the problem.

Semaphores may also be used to interlock user threads. To illustrate the use of semaphores we have recoded the multiplexer example presented previously to use multiple threads interlocked by a semaphore in place of the `alt` functions.

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

#define STACKSIZE 1024

CHAN chan, consumer_finished;

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

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

main()
{
    int dummy;

    chan_init(&consumer_finished);
    chan_init(&chan); /* BEFORE starting the threads! */

    thread_create(producer, STACKSIZE, 0);
    thread_create(consumer, STACKSIZE, 0);

    /* wait for all threads to terminate */
    chan_in_word(&dummy, &consumer_finished);

    /* before exiting */
}
```

Figure 5.6: Synchronization by Internal Channels

A new execution thread is created for each input port. Each thread does a simple sequential read and waits for a message. As soon as one thread receives a message it waits until a semaphore indicates the output port is free. 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 5.7 shows the semaphore version of the multiplexer 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 [7, 8]. 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.

### 5.6.2 Threads versus Tasks

Threads can be useful in many situations. They are just “lightweight” processes, corresponding to processes in Modula-2 or the coroutines 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;

```

/* mux.c:    message multiplexer 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(int argc, char *argv[], char *envp[],
     CHAN *in_ports[], int ins, CHAN *out_ports[], int outs)
{
    extern void receive(int);
    int i;

    sema_init(&buf_free, 1);     /* buffer is initially free */

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

    for (i=0; i < ins; i++)      /* one 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(int i)              /* handle a single input port */
                                /* i = 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,        /* read body of message into */
                         &buf[0],       /* the shared global buffer */
                         in_p[i] );     /* 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 5.7: Multiplexer Task Using Semaphores

- they can share data and may communicate either via shared memory or by using channels like tasks;
- 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.

## 5.7 Debugging

This section contains some hints on debugging parallel programs for users who have not purchased Tbug, 3L's interactive debugger product. Tbug allows examination of source program variables and provides source level breakpointing and single-stepping within multi-threaded, multi-tasking Parallel C applications. This simplifies fault-finding.

If you do not have Tbug, 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 **afserver**. 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 stand-alone 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 stand-alone 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 stand-alone 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 **afserver** in the second PC 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_PC type=PC
processor extra_B004             !plugged into extra_PC
task extra_afserver ins=1 outs=1
wire ? extra_B004[0] extra_PC[0]
wire ? extra_B004[1] root[1]

place extra_afserver extra_PC
place suspect_task extra_B004

connect ? suspect_task[1] extra_afserver[0]
connect ? extra_afserver[0] suspect_task[1]

```

The main thing to notice here is the `type=PC` attribute given to the `extra_PC` 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 `afserver` on the extra PC using the `afserver` command *without* the `-:b` option before starting the system under test. If no `-:b` option is present on the command line, the `afserver` 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 PC with an Inmos link adapter card. Relink the suspect task with the full run-time library rather than the stand-alone library, then re-configure the system with input and output ports 1 of the task being debugged connected to the PC with the link adapter, as follows:

```

processor second_PC type=pc
task second_afserver ins=1 outs=1
place second_afserver second_PC

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

task suspect_task ins=2 outs=2 !connect [1]'s to afserver
place suspect_task any_processor

```



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

This technique has two advantages: it only requires an extra PC and link adapter card, rather than an extra PC 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 PC to accept debugging messages from up to four separate tasks anywhere in the network being debugged and multiplex them onto its PC screen.

## 5.8 Estimating Memory Requirements

Section 3.5 has already discussed the various categories of data storage. As noted there, the data requirement for a task is the sum of the number of bytes required for static, stack and heap storage in all its modules.

The **decode** utility (see chapter 14) 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 (such as `fprintf` or `printf`) 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 18.

## Chapter 6

# Global Input/Output

In the last chapter, we looked at how to build configured applications with more than one user task, whether running on one or more transputers. In this chapter, we shall see how to arrange for all these tasks to use the input/output functions and other facilities which need the support of the `afserver` program.

### 6.1 One Transputer

We saw in section 5.3 that only one task can communicate with the `afserver`, and that this task was the only one to be linked with the full C run-time library. All the other tasks were linked with the stand-alone library, and this precluded them from doing I/O, DOS calls and so on. Figure 6.1 shows, for example, a simple two-task application, and figure 6.2 shows the corresponding configuration file.

The problem is that the server only has one possible connection to one filter task, and the filter task has only one possible connection to a user task. We can get round this problem by placing a special multiplexer task between the user tasks and the filter tasks. This

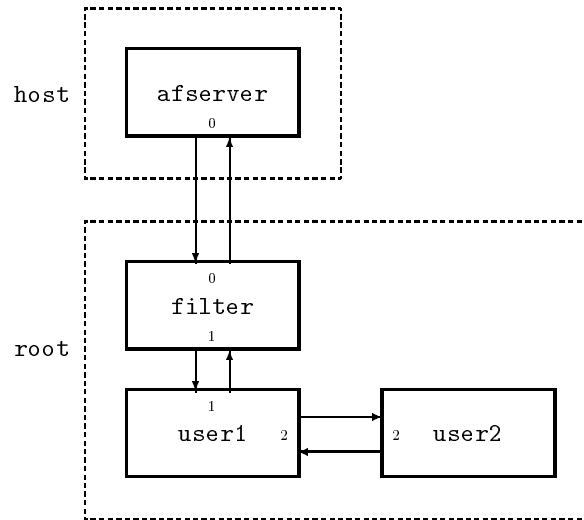


Figure 6.1: Two-task Application

multiplexer task is included with the Parallel C kit, and is called **filemux**; a task data sheet for it can be found in chapter 20.

Figure 6.3 shows this arrangement. The configuration file is unchanged, except that the following statements, which connected **user1** to the filter are removed:

```
connect ? filter[1] user1[1]
connect ? user1[1] filter[1]
```

and instead we have the following:

```
task filemux ins=3 outs=3 data=6.5K
place filemux root
connect ? filter[1] filemux[0]
connect ? filemux[0] filter[1]
connect ? filemux[1] user1[1]
connect ? user1[1] filemux[1]
connect ? filemux[2] user2[1]
connect ? user2[1] filemux[2]
```

Now it is **filemux** which is connected to the filter, and the two user

```

processor host
processor root
wire ? root[0] host[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task user1 ins=3 outs=3 data=50K
task user2 ins=3 outs=3 data=50K
place afserver host
place filter root
place user1 root
place user2 root
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] user1[1]
connect ? user1[1] filter[1]
connect ? user1[2] user2[2]
connect ? user2[2] user1[2]

```

Figure 6.2: Two-task Application

tasks each have their number 1 port pairs connected to a **filemux** port pair. Each user task should be linked with the full run-time library using **t4ctask** or **t8ctask**, and each task can behave as if it has sole use of the **afserver**. The multiplexer arranges for all the messages from the user tasks to be transported to the **afserver** on the host, and transports the replies back to the correct user task.

You can arrange for the multiplexer to handle more tasks. Each must have its port pair 1 connected to a multiplexer port pair, starting at number 1 and going upwards with no gaps. For example, if the multiplexer is supporting 9 tasks, they must be connected to port pairs 1 to 9. The amount of memory which the multiplexer uses is no more than  $(6 + 0.25n)$ K bytes, where  $n$  is the number of tasks supported. So in the case of 9 supported tasks, the TASK statement should read:

```
task filemux ins=10 outs=10 data=8.25K
```

The multiplexer adjusts its own activities to support all the tasks which are connected in this way.

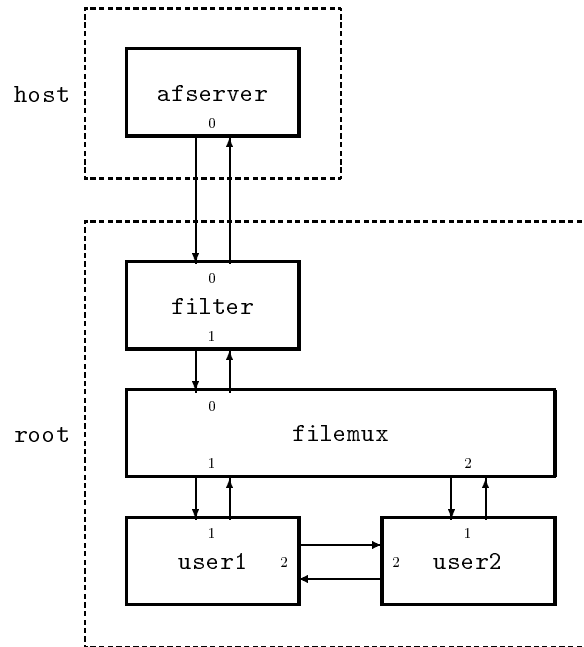


Figure 6.3: Two-task Application with Global I/O

## 6.2 More than One Transputer

A task does not have to be on the same transputer as the multiplexer which supports it. Provided the necessary wires exist, it can be on an adjacent transputer. Figure 6.4 shows how this would be arranged, and figure 6.5 is the corresponding configuration file.

Each WIRE statement corresponds to a hardware link between two transputers, and supports two CONNECT statements, one in each direction. This means that the connections between `filemux` and one supported task on a neighbouring transputer will use up one WIRE statement, that is, one hardware link. This implies two restrictions:

- If you have a task on a neighbouring transputer supported by a multiplexer on this one, and you also want user tasks on the

two transputers to be connected, you will need *two* hardware links between the two transputers.

- As a transputer has only four hardware links, the number of tasks on neighbouring transputers which can be supported is limited.

## 6.3 More than One Multiplexer

Fortunately, there is a way to improve on this situation. This can be done by using more than one copy of the **filemux** task.

Up to now, the number 0 port pair of the multiplexer has always been connected to the number 1 port pair of the filter task. However, it is also possible to connect the number 0 port pair to another copy of the multiplexer, which could be on another transputer. In this way, copies of the multiplexer can be built up into a tree. Figure 6.6 shows how this could be done, and figure 6.7 shows the corresponding configuration file.

Once again, a user task which is connected to the multiplexer, no matter how deep into the tree it is, can use the server's facilities as if it were directly connected. The task's server requests are passed up the tree of multiplexer tasks until they reach the **afserver**, and the response is similarly passed back to the correct user task.

## 6.4 Limits

The number of MS-DOS files and devices which the **afserver** can handle at the same time is limited, currently to 20. This means that the network of tasks which are supported by **filemux** may not open more than 20 files at any one time. This applies regardless of the number of **filemux** tasks involved.

Each C task which is linked with the full run-time library uses up three of this allotment of 20, for `stdin`, `stdout` and `stderr`. As a result, the maximum number of tasks which can be supported by the multiplexer network is currently 6.

## 6.5 Termination of an Application

When a task which is linked to the full run-time library terminates, for example by returning from the `main` function or calling the `exit` function, it sends to the `afserver` a *server terminate* request. This causes the `afserver` to stop executing and return control to DOS.

Obviously, when a number of tasks are using the server, this cannot be allowed to happen. Accordingly, `filemux` does not pass on a server terminate request until all the tasks it supports have tried to send one.

The effect of this is that the `afserver` does not terminate until it has been asked to do so by every task in the application which is supported by `filemux`. It is not enough for a task to go into a loop, or to be waiting for input; if this happens, the application as a whole will not terminate. Every task must terminate properly.



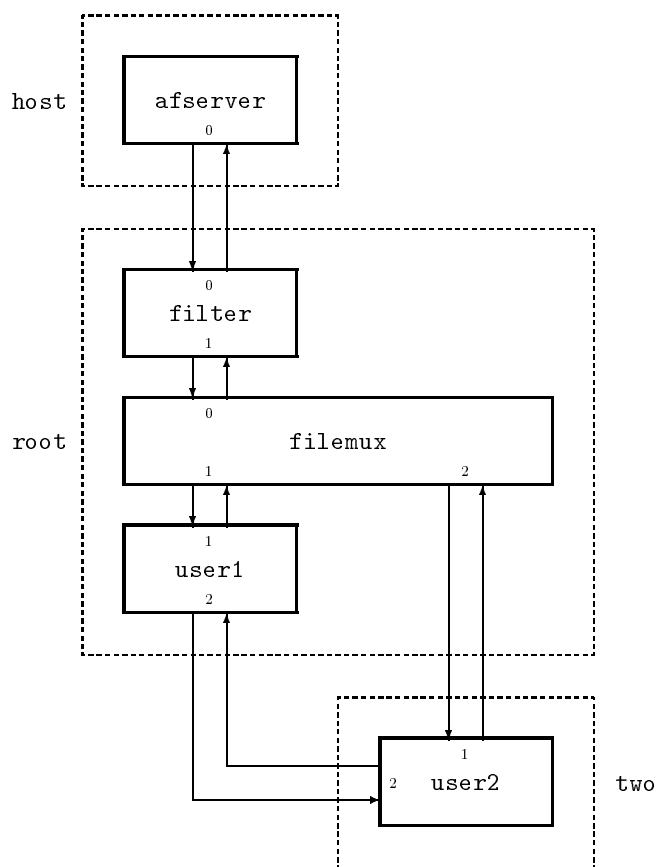


Figure 6.4: Task on Neighbouring Transputer

```
processor host
processor root
processor two
wire ? root[0] host[0]
wire ? root[1] two[0]
wire ? root[2] two[1]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task filemux ins=3 outs=3 data=6656
task user1 ins=3 outs=3 data=50K
task user2 ins=3 outs=3 data=50K
place afserver host
place filter root
place filemux root
place user1 root
place user2 two
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] filemux[0]
connect ? filemux[0] filter[1]
connect ? filemux[1] user1[1]
connect ? user1[1] filemux[1]
connect ? filemux[2] user2[1]
connect ? user2[1] filemux[2]
connect ? user1[2] user2[2]
connect ? user2[2] user1[2]
```

Figure 6.5: Task on Neighbouring Transputer

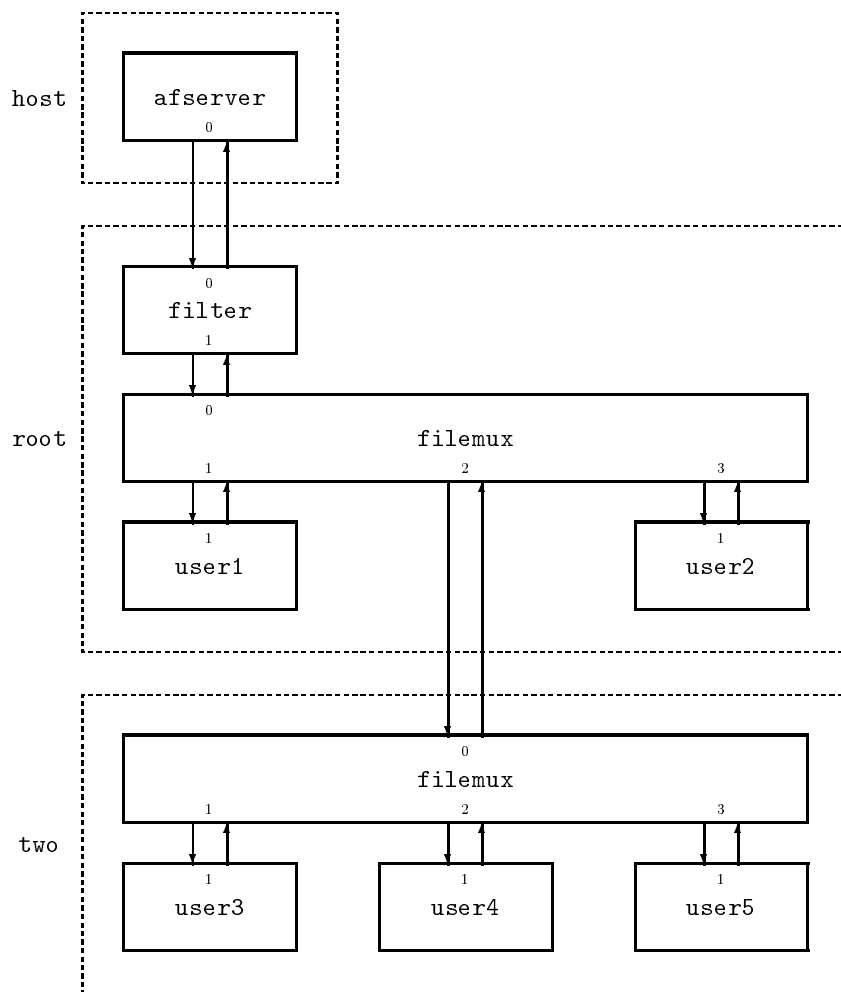


Figure 6.6: Networking Multiplexers

```

processor host
processor root
processor two
wire ? root[0] host[0]
wire ? root[1] two[0]

task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task mux1 file=filemux ins=4 outs=4 data=6912
task mux2 file=filemux ins=4 outs=4 data=6912
task user1 ins=2 outs=2 data=50K
task user2 ins=2 outs=2 data=50K
task user3 ins=2 outs=2 data=50K
task user4 ins=2 outs=2 data=50K
task user5 ins=2 outs=2 data=50K
place afserver host
place filter root
place mux1 root
place mux2 two
place user1 root
place user2 root
place user3 two
place user4 two
place user5 two
connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] mux1[0]
connect ? mux1[0] filter[1]
connect ? mux1[1] user1[1]
connect ? user1[1] mux1[1]
connect ? mux1[3] user2[1]
connect ? user2[1] mux1[3]
connect ? mux1[2] mux2[0]
connect ? mux2[0] mux1[2]
connect ? mux2[1] user3[1]
connect ? user3[1] mux2[1]
connect ? mux2[2] user4[1]
connect ? user4[1] mux2[2]
connect ? mux2[3] user5[1]
connect ? user5[1] mux2[3]

```

Figure 6.7: Networking Multiplexers

## Chapter 7

# Processor Farms

The previous chapters 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 4 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 displays pictures of the now-famous “Mandelbrot Set” on an IBM PC-type host equipped with a CGA-compatible display as an example processor farm application.

The full source code of the Mandelbrot master and worker tasks, and of the configuration file required, is printed in appendix G. These files are also supplied in machine-readable form in the `\tc2v2\examples` directory, along with a batch files (`mandelt4.bat` and `mandelt8.bat`) to compile, link and configure the example files into an executable application. Section 7.5 at the end of this chapter explains how to run the demonstration if you want to try it out before reading further.

The Mandelbrot program is suitable for running on a processor farm because each part of the final picture 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 Mandelbrot case this is easy: the master divides up the screen area into 100 small squares, and sends the coordinates of the individual squares out into the network as work packets. Any idle worker receiving a packet calculates the required graphics display bitmap for that part of the picture and sends it back as a result packet.

Both the master and worker task make use of a package of functions (the “`net`” functions) which 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. 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_send` and `net_receive` functions are described in detail in the reference part of this manual, starting on page 290.

## 7.1 The Worker Task

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

1. Get a work packet by calling `net_receive`. The work packet identifies the individual square of the display which is to be computed.
2. Work out the graphics display for that square in the `counts` array member of the result packet structure `r`.
3. Send the result packet back to the master task by calling `net_send`.
4. Go back to step 1.

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 `afserver` it must be linked with the stand-alone run-time library.

## 7.2 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. Because the routing software is buffered, the network can hold a number of packets waiting to be processed; this ensures that processors are idle for a short a time as possible. Consequently, the network will not be saturated until all the workers are working, and all the buffers are full.

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 by 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 Mandelbrot program these functions are performed by three parallel execution threads: `send`, `receive` and `main`, which are synchronized using semaphores.

## 7.3 The net Package

Descriptions of the functions in the `net` package may be found in chapter 11.

The administration of a processor farm is under the control of a task called `frouter` (see chapter 20). Each node in a processor farm contains a copy of this task; all the copies, and the master and worker tasks, are connected together by the flood-filling configurer (see section 7.4 below). This network of `frouter` tasks can be regarded by the programmer as a single entity, whose job it is to ensure that messages arrive at their correct destinations.



### 7.3.1 Functions `net_send` and `net_receive`

The function `net_send` is used to send a message to the network, and `net_receive` is used to receive one from the network.

Messages sent to the network by the master task (using `net_send`) are routed to an idle worker task, if necessary passing through more than one node in order to reach one. At each level of re-direction, the messages are buffered. Only if all the worker tasks are busy, and all the buffering is full, will a call on `net_send` by the master task have to wait.

Messages sent to the network by worker tasks are routed back to the master task, once again passing through more than one transputer if necessary.

There is a limit on the size of a buffer that can be submitted to `net_send`; the constant `NET_MAX_PACKET_LENGTH` is defined in the package file to have this value (currently 1024). If the message you wish to send is longer than this, it must be broken into a number of *packets*. The last packet of the message should be sent with the `complete` parameter of `net_send` set to the value 1; this should also be done if there is only one packet in the message. All the other packets should be sent with `complete` set to the value 0. When a packet is received, `net_receive` sets its `complete` parameter to the value used when the packet was sent. The network will ensure that a sequence of packets will arrive in the right order, but it is the receiving task's responsibility to fit the sequence of packets back together again.

It is best, however, to design the application to use messages which are smaller than 1024 bytes, as long packets can clog up the network and block packets being delivered to other nodes.

### 7.3.2 The `net_broadcast` function

Sometimes you may wish to start a run of your processor farm application by initialising all the worker tasks with the same set of data. These could be parameters obtained from the user, for example, or data tables which vary from run to run. This can be done using `net_broadcast`.

The `net_broadcast` function should only be used by the master task. Each call results in a copy of the broadcast message being sent to every worker task in the processor farm. The broadcast message can be received by the worker tasks by using `net_receive` in the normal way. The most usual time to do a broadcast would be at the beginning of the run, but a message can be broadcast whenever the network is idle; that is, when all the work packets sent out by the master task have been answered by the worker tasks by sending a result packet. However, as there is no method to tell a broadcast message from a normal work packet, it is up to the programmer to ensure that the worker tasks never get confused.

A broadcast message can be any length. If necessary, `net_broadcast` will break it up into packets for transmission through the network. In this case, the worker tasks will have to call `net_receive` more than once to receive it, checking the `complete` parameter as described above.

Note that `net_broadcast` is the only reliable method to send an identical message to every worker task. Repeatedly calling `net_send` is unlikely to work.

## 7.4 Building the Application

Once the master and worker tasks have been compiled, the master should be linked with the standard run-time library (`t4ctask` or `t8ctask`); the worker task must be linked with the stand-alone run-time library (`t4cstask` or `t8cstask`).

The executable file containing the code of these tasks along with the extra software to flood-fill a transputer network with copies of the worker task is generated by the flood-fill configurer, `fconfig`.

### 7.4.1 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 Mandelbrot 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 were used, the configurer would expect to find the tasks in files called `mandelm.b4` and `mandelw.b4`.

```
task master file=mandelm
task worker file=mandelw 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.

It is not required (and indeed not possible) to specify INS or OUTS attributes for the master and worker tasks in their configuration TASK statements: all the ports and connections required are generated automatically by the flood-configurer.

To run the flood-configurer by hand, use a command of the form:

```
fconfig configuration-file executable-file
```

For example:

```
C>fconfig mandel.cfg mandel.app
```

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 **afserver** 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 the current version 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 and filter tasks as well.

## 7.5 Running the Example

The kit includes batch files which will automatically compile, link and configure the the Mandelbrot example.

To run the program from a temporary directory on a network of T8s, you can use the following commands:

```
C>cd \  
  
C>mkdir temp  
  
C>cd temp  
  
C>copy \tc2v2\examples\*.*  
  
C>mandelt8  
:  
:
```

To run on a network of T4s, you should use the command `mandelt4` instead of `mandelt8`. Each of these batch files results in an application file (called `fmandel.b4`) which can be run in any network consisting only of the appropriate type of transputer. Section 7.6 below describes how to flood-configure applications to run on a network containing a mixture of T4 and T8 processors.

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

```
C>afserver -:b fmandel.b4
```

When it starts, the Mandelbrot program reminds you that it needs an IBM PC compatible host machine with CGA graphics to work properly, then prompts you to enter several numeric parameter values on the keyboard.

Some suitable test values are:

```
Input X coordinate: -2  
Input Y coordinate: -1.25  
Input Y range:      2.5  
Threshold 1: 5  
Threshold 2: 20  
Threshold 3: 50
```

Once the display is complete, the host system's bell will be rung. Hit Enter, and the first prompt will reappear. You can then experiment

with other sets of parameter values. A more interesting set of values is:  $-0.25$ ,  $0.8$ ,  $0.25$ ,  $10$ ,  $20$ ,  $50$ .

Use Ctrl-C when you want to stop the program.

Once you have the program working, you can make it run faster simply by plugging more transputers into the network and rebooting the program.

The batch files `mandelt4.bat` and `mandelt8.bat` also result in another file, `mandel.b4`. This is a statically configured application including a master task and one worker, which are both placed on the root transputer.

## 7.6 Heterogeneous Networks

A flood-filled application compiled for the T4 and configured using the simple `master` and `worker` forms of task declaration may work on a mixed network of T4 and T8 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 T4 and T8 instruction sets.

Mixed networks of T4 and T8 processors are properly handled by an extension to the configuration file, like this:

```
task t4master file=mandelm4
task t8master file=mandelm8
task t4worker file=mandelw4 data=10k
task t8worker file=mandelw8 data=10k opt=stack
```

Separate tasks must be compiled and linked for T4 and T8 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 T4 and T8 variants of a task. This is useful because the T4 and T8 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.

At present, T425 processors cannot be included in heterogeneous flood-filled networks.





## Chapter 8

# Developing T2 Programs

This chapter shows you how to use the Parallel C compiler to develop programs for 16-bit transputers, hereafter referred to as T2 transputers. Many of the features of the compiler are the same whether you are compiling for T2, T4 or T8 transputers.

The preceding chapters describe working with T4 and T8 transputers in detail. This chapter will concentrate on the differences you will see between T2 transputers and the other variants. For the convenience of T2 users, appropriate information drawn from chapters 9, 10, 11 and 12 is also be presented here.

### 8.1 Compiling

T2 support

A C source program is compiled into a binary object (`.bin`) file of T2 transputer instructions by a command of the form:

```
t2c source-file
```

Note that code compiled for a T2 will not run on 32-bit transputers (and vice versa) so you must use the command appropriate for the type of processor you have in mind.

The *source-file* is the filename of the C source program which is to be compiled. If no filename extension is given in the command, `.c` is added automatically.

So, to compile the file `hello.c` for the T2, you would give the command:

```
C>t2c hello
```

If the source file contains no errors, an output file `hello.bin` is produced. If the compiler detects errors in the source program, it writes diagnostic messages to the MS-DOS standard output stream.

## 8.2 The Compiler in T2 Mode

### 8.2.1 Language Restrictions

This section should be read in conjunction with section 9.1.

The compiler imposes a number of restrictions on the sorts of program it can handle when generating instructions for T2 transputers.

- There is no support for floating-point quantities. This means that you cannot declare variables of type `float` or `double`. Similarly, you cannot use floating-point constants.
- The only sizes of integer variable you can use are `int` (signed or unsigned), `short`, and `char`. `int` and `short` will give you a 16-bit integer which will have an *even* address. `char` will give you an 8-bit unsigned integer.
- Any integer expressions which are evaluated while the program is being compiled (for example, values used for conditional

compilation with `#if`) will be evaluated to 32-bit accuracy. This can give different answers from the evaluation of similar expressions during program execution. For example, the condition on the following `#if` statement is true:

```
#if (0x8000 << 1) >> 1 == 0x8000
```

However, note the effect of the following expressions when the program runs on a T2 transputer:

```
int x,y,z;
x = 0x8000;
y = x<<1;      /* y will get the value 0 */
z = y>>1;      /* z will get the value 0 */
```

- Integer constants to be used during program execution will be evaluated to 32-bit accuracy and the least significant 16 bits will be used. A warning will be issued if the most significant 16 bits of the 32-bit representation are not all the same. This means that, for example, if you write the constant `0x12345` in a program, the compiler will generate a warning message and use the value `0x2345` instead.
- If you use the keyword `long` in a declaration, the compiler will ignore it and warn you of this fact.

### 8.2.2 Pre-defined Macros

When compiling for T2 transputers, the following macros will be automatically defined with the value '1':

```
_transputer
_3L
_IMST2
```

The following macros will be undefined, as they are used to indicate that a T4 or T8 compilation is being performed:

```
_IMST4
_IMST8
_IMST8A
```

See also section 9.5.

### 8.2.3 Data-type Representations

A full discussion of data-type representations for all processors may be found in section 9.8.

#### 8.2.3.1 Integral Data Types

On the T2, a byte is 8 bits and a word is 16 bits (2 bytes). The C integral (i.e., integer or character) data types are represented as follows:

Type	Bits	Bytes	Minimum	Maximum
char	8	1	0	255
signed char	8	1	-128	127
int	16	2	-32768	32767
unsigned int	16	2	0	65535
short int	16	2	-32768	32767
unsigned short int	16	2	0	65535

The `long` data types are not supported.

#### 8.2.3.2 Pointer Types

All pointer types (i.e., types of the form “pointer to  $x$ ”) are represented by a single word (2 bytes, 16 bits) whose value is the address of the object pointed to.

#### 8.2.3.3 Floating Types

Floating types are not supported on the T2.

## 8.2.4 Compiler Error Messages

The following special messages may be output when the compiler is working on a T2 program.

- no support for "double" types
- no support for "float" types
- no support for "long" types
- Warning: integer constant truncated to 16 bits at line *number*

## 8.3 Linking T2 Tasks

Once a C program has been compiled into an object (`.bin`) file, it must be linked with any external functions it requires before it can be run. This operation is performed by the linker, `linkt`.

Standard functions are provided in `sacrtlt2.bin`, the T2 stand-alone run-time library. A single batch file is provided to link together as many as nine object files with this library to produce an executable (position-dependent) program.

To link a single T2 object file produced with the `t2c` compiler use the command:

```
t2clink object-file
```

This is equivalent to the command:

```
linkt/m64k object-file sacrtlt2
```

To link multiple T2 object files use the command:

```
t2clink object-file1 object-file2 ...
```

For example:

```
C>t2clink main bits pieces
```

The `t2clink` batch file assumes that the target T2 processor has 64KB of read-write memory. If this is not the case with your processor, you must add appropriate switches to the command line, as described below.

## 8.4 Linker Support for the T2

The linker is described in full in chapter 12. This section covers only those facilities provided to support the T2 processors.

### 8.4.1 Linker Command Switches

The following command-line switches are *only* for use when linking code for T2 processors:

<code>/Msize</code>	define size of read-write memory area (including on-chip memory)
<code>/Asize</code>	define size of stack area
<code>/FC</code>	optimise code area
<code>/FA</code>	optimise stack (automatic) area
<code>/FS</code>	optimise static data area
<code>/FH</code>	optimise heap area
<code>/Rsize</code>	define size of read-only memory area

When you give the size of an area you can specify it either in bytes (e.g., 4096) or in kilobytes (e.g., 4K).

#### 8.4.1.1 Switch `/Msize`

This switch gives the total number of bytes of read-write memory available to the program. The memory will be used to hold the static data, heap and stack for the running program. In addition, it will

hold the executable code of the program unless the code is to be held in read-only memory.

You must give a `/M` switch when linking for T2 systems unless you intend to control the linker's memory allocation by means of modified `/F` switches.

The batch file `t2clink` provides a default value of 64K for this switch (`/M64K`) but you may override this default with another `/M` switch of your own, e.g.,

```
C>t2clink main bits pieces/m24k
```

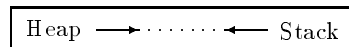
If more than one `/M` switch appears on a command line, only the last will have any effect.

The linker will give a warning if you specify less than 2048 bytes (the size of the on-chip RAM) or more than 65536 bytes of read-write memory.

#### 8.4.1.2 Switch `/Asize`

This switch controls the number of bytes of read-write memory to be used for the stack (“automatic” storage in C terminology). The linker will give a warning if you specify less than 128 bytes of stack. Memory for the stack is taken from the read-write memory remaining after the code and static data areas have been allocated.

If you do not specify this switch then the whole of the remaining memory will be used for a combined heap and stack area. The stack will grow towards the heap from the more positive end of the area while the heap will grow towards the stack from the more negative end of the area.



If you do give a `/A` switch, the given amount of memory will be allocated to the stack and the whole of the remaining memory will

be used for the heap. In this case the stack and heap areas will be considered distinct and will not interact.

#### 8.4.1.3 Switches `/FC`, `/FA`, `/FS`, and `/FH`

These switches are used to control the order in which the various areas of the program are loaded into the available memory: `/FC` for code, `/FA` for the stack (automatic) area, `/FS` for static data, and `/FH` for the heap.

The linker will usually construct an executable image by laying out the various areas (code, static data, heap, and stack) in memory, starting at the most negative address usable—in the fast, on-chip memory. Consequently the parts of the image which are placed first will benefit from the speed of this memory.

The `/F` switches give you control over the order in which the areas will be laid out. Any area mentioned in a `/F` switch will be considered a candidate for “optimisation”—you can think of the ‘F’ as standing for “fast”. For example, the switches `/FC/FS` indicate that the code and static data areas are to be optimised. The order in which you give the `/F` switches is of no significance.

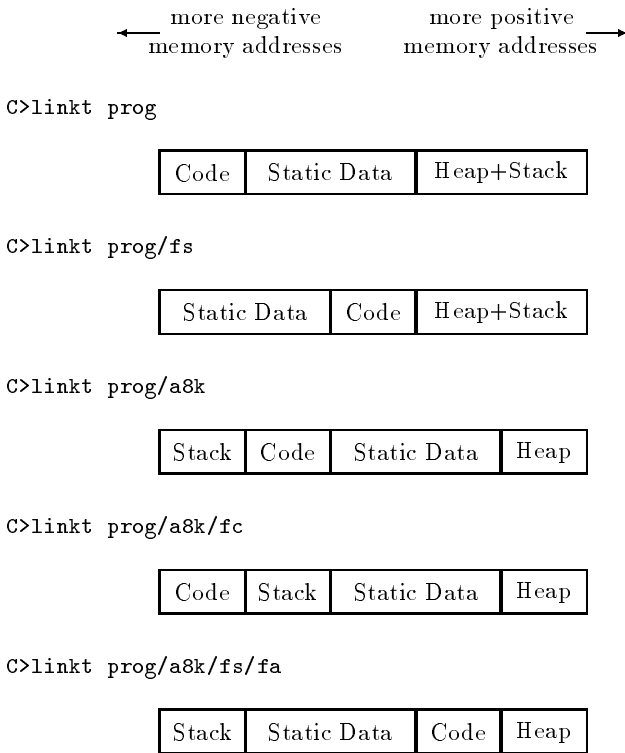
The linker will lay out all of the optimised areas before it lays out any non-optimised areas. The order in which areas (optimised or not) are laid out depends on the presence or absence of the `/A` switch.

If you do not specify the `/A` switch, then the stack and heap areas will be combined, as described above. In this case the linker will lay out the areas in the order: code, static data, and then the combined stack and heap.

If you do specify the `/A` switch, then the stack and heap areas will remain distinct, and the linker will lay out the areas in the order: stack, code, static data, and then heap.



The following pictures should clarify this procedure. Note that in these pictures addresses grow more positive towards the right hand side.



The system described is designed to allow the most common requirements to be specified simply.

#### 8.4.1.4 Modified /F Switches

The /F switches may be modified so that instead of simply marking areas for optimisation they explicitly specify the memory locations to be used.

To modify the switches you append an address and size specification of the form *start:size*, where *start* is the address for the start

(smallest address) of the area and *size* is the size of the area in bytes. If *start* or *size* begin with a ‘#’ character they will be interpreted as hexadecimal, otherwise they will be interpreted as decimal. All values of *start* and *size* must be even. Note that the start address of the stack area is *not* the initial value for Wptr; that value will be *start + size*. For example:

```
C>linkt x y z /fc#1000:80 /fh#2000:#2000 /fa#8000:4096 /fs0:8
```

The linker will check that these areas do not overlap and issue an error message if they do. Similarly, the linker will issue an error message if either the code area or the static data area is too small for the linked image. The total size of the static data area for a task will be:

$$2 \times \text{modules} + \sum_{i=1}^{\text{modules}} \text{static}_i$$

There are several implications of modifying /F switches in this way:

1. If you specify one modified /F switch then you must specify and modify all four. The only exception to this rule is when you are linking for ROM (described later);
2. There will be no automatic optimisation or memory allocation. Memory allocation is fully under your control;
3. The stack and heap areas will be considered separate, even though they may be adjacent. This means that while the program is running the heap will never extend into the stack area.

#### 8.4.1.5 Switch /Rsize

The /R switch instructs the linker to generate an image suitable for burning into ROM. The image size will be exactly the number of bytes specified in the /R switch.

When a ROM program starts execution, it copies its static data from the ROM into read-write memory.

The code may either be left in the ROM or copied into the read-write memory. This is controlled by the `/FC` switch. If no `/FC` switch is specified then the code will be executed from the ROM. If `/FC` is specified (modified or not) then the code will be copied into the read-write memory before being executed.

Note that when linking for ROM with modified `/F` switches you may omit the modified `/FC` switch if you wish the code to be executed from ROM. Of course, you should make sure that none of the areas overlaps any ROM addresses.

### 8.4.2 The Bootstrap

Programs are loaded into T2 systems by a special piece of code called the “bootstrap”. This code needs to use about 160 bytes of read-write memory while it is loading your program. The linker will automatically arrange for the bootstrap to use part of the memory that eventually will be used for your stack area or heap area. You will get a fatal error from the linker if it finds that neither of these areas is large enough for the bootstrap, so you should ensure that one or other of them is at least 160 bytes in size.

## 8.5 The Run-Time Library

The T2 is supported only by a stand-alone run-time library, `sacrtlt2.bin`. This contains a subset of the functions described in chapters 10 and 11.

As we noted above, a word on the T2 is 16 bits (2 bytes). This means that functions such as `chan_in_word` will only transfer 2 bytes of data.

The following sections list the functions that are defined in the specified header files. Discussions of each of the various modules of the run-time library may be found in chapter 10, and each individual function is described in chapter 11, which is arranged alphabetically by function name.

### 8.5.1 Functions Defined in `alt.h`

```
alt_nowait      alt_nowait_vec  alt_wait
alt_wait_vec
```

### 8.5.2 Functions Defined in `chan.h`

```
chan_in_byte      chan_in_byte_t    chan_in_message
chan_in_message_t  chan_in_word      chan_in_word_t
chan_init         chan_out_byte      chan_out_byte_t
chan_out_message  chan_out_message_t chan_out_word
chan_out_word_t   chan_reset
```

### 8.5.3 Functions Defined in `chanio.h`

```
_inmess  _outbyte  _outmess
_outword
```

### 8.5.4 Functions Defined in `ctype.h`

```
isalnum  isalpha  isascii
iscntrl  isdigit  isgraph
islower  isprint  ispunct
isspace  isupper  isxdigit
tolower  toupper
```

### 8.5.5 Functions Defined in locale.h

localeconv setlocale

### 8.5.6 Functions Defined in par.h

par\_free par\_malloc

### 8.5.7 Functions Defined in sema.h

sema\_init        sema\_signal   sema\_signal\_n  
sema\_test\_wait   sema\_wait     sema\_wait\_n

### 8.5.8 Functions Defined in setjmp.h

longjmp setjmp

### 8.5.9 Functions Defined in signal.h

raise signal

### 8.5.10 Functions Defined in stdlib.h

abs            atexit   atoi  
bsearch       calloc   cfree  
div           exit     free  
malloc        mblen    mbstowcs  
mbtowc       qsort    realloc  
wcstombs     wctomb

### 8.5.11 Functions Defined in `string.h`

<code>memchr</code>	<code>memcmp</code>	<code>memcpy</code>
<code>memmove</code>	<code>memset</code>	<code>strcat</code>
<code>strchr</code>	<code>strcmp</code>	<code>strcpy</code>
<code>strcspn</code>	<code>strlen</code>	<code>strncat</code>
<code>strncmp</code>	<code>strncpy</code>	<code>strpbrk</code>
<code>strrchr</code>	<code>strspn</code>	<code>strstr</code>
<code>strtok</code>		

### 8.5.12 Functions Defined in `thread.h`

<code>thread_create</code>	<code>thread_deschedule</code>	<code>thread_priority</code>
<code>thread_restart</code>	<code>thread_start</code>	<code>thread_stop</code>

### 8.5.13 Functions Defined in `timer.h`

<code>timer_after</code>	<code>timer_delay</code>	<code>timer_now</code>
<code>timer_wait</code>		

## 8.6 Running T2 Programs

Usually you would run T2 programs under the control of another transputer (or completely stand-alone, e.g. from ROM). It is unlikely that you would be able to run a T2 program directly from the server running in the host.

There are two ways in which the T2 shown in figure 8.1 can be loaded. The first, and preferred, method is to use the general configurer and the second method is to use a small program on the T8 to pass the code through to the T2. These methods are described in the following sections.

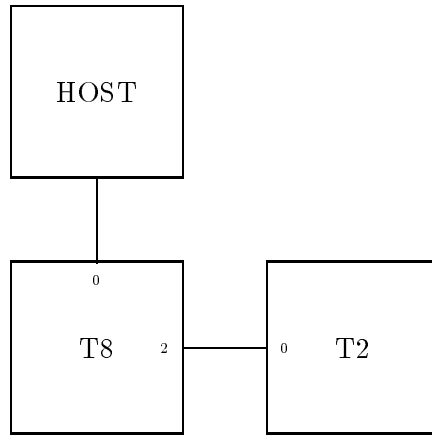


Figure 8.1: Example Network with T2

### 8.6.1 Using the Configurer to Boot a T2

The use of the general configurer, `config`, is discussed in chapter 5. Details of the configuration language, and more about the functioning of the configurer may be found in chapter 18.

The configurer can be used to boot a processor at the edge of a network with code from a specified file. Processors using this mechanism are declared in the configuration file using the `BOOT` attribute, which is described in more detail in section 18.2.6.1.

The configuration file should describe the main network of T4 and T8 transputers. The T2 processor should be declared using the `BOOT` attribute to specify the file which contains the code for the T2.

The wire between the T2 and the main network should be declared.

The task in the main network which will communicate with the T2 task should have its ports bound to the appropriate link addresses. You must use the actual hardware link addresses to do this.

For example, the main network in figure 8.1 consists of a single T8

so the configuration file could be as follows:

```

! Example of configuring with T2
processor host
processor root                ! T8
processor P001 B00T="t2code.b4" ! T2

wire ? host[0] root[0]      ! connect PC to network
wire ? root[2] P001[0]

! Task declarations
task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task monitor ins=3 outs=3

! Assign software tasks to physical processors
place afserver host
place filter root
place monitor root

! Set up the connections between the tasks.
connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]

connect ? filter[1] monitor[1]
connect ? monitor[1] filter[1]

! bind ports to link to T2 processor
bind input  monitor[2] value=&80000018 ! I/O to T2
bind output monitor[2] value=&80000008 ! over link 2

```

Here, `monitor` is a task compiled and linked for the T8 as described in chapter 5. The `monitor` task would communicate with the task on the T2 using its port pair 2. The T2 processor would be booted with the code from the file `t2code.b4`.

### 8.6.2 Piping Code into a T2

It may be preferable to use a program on the T8 to pass the code through to the target T2. This mechanism is only suitable when the target T2 is connected directly to the root transputer.



The program below shows how the T2 in figure 8.1 can be loaded from the T8. It passes the code of the file named as a parameter to output link 2 and then calls a function which the user would write to communicate with the program in the T2.

```
#define TIMEOUT 31250          /* about 2 seconds */
#include <stdio.h>
#include <chan.h>

main(int argc, char *argv[])
{
    FILE *t2f;
    int n, e;
    char buffer[256];
    if (argc != 2) {
        printf("T8: Wrong number of arguments\n"); return;
    }
    /*
    ** open the T2 image file
    */
    if ( (t2f = fopen(argv[1], "rb") == NULL) {
        printf("Cannot access %s\n", argv[1]); return;
    }
    /*
    ** boot the image down link 2
    */
    while (n = fread(buffer, 1, sizeof(buffer), t2f)) {
        if (!chan_out_message_t(n, buffer, Link2Output, TIMEOUT)) {
            printf("Time-out while booting T2\n"); return;
        }
    }
    /*
    ** communicate with the T2
    */
    User_Function();
}
```

This program would be compiled and linked for the T8 and then run as follows:

```
C>afserver -:b progrname.b4 codefile.b4
```

Here, *progrname* is the pipe program listed above and *codefile* is the name of the file of code compiled and linked for T2 transputers.

## 8.7 Parameters to Main

A T2 program will be invoked from the surrounding initialisation code by a call to the function `main`. The call will pass in two channel pointer parameters `boot_link_in` and `boot_link_out`. These parameters will contain respectively the addresses of the input link from which the program was loaded and its corresponding output link address.

```
void main(CHAN *boot_link_in, CHAN *boot_link_out)
{
    ...
}
```

If the T2 program has been loaded from ROM then both of these parameters will have the value zero.

Note that the parameters passed to a T2 `main` function are quite different from those passed to a T4 or T8 `main` function, as described in section 9.2.

# Introduction

## Overview

The chapters which follow provide detailed reference material for use with the Parallel C compiler. They are intended for use by readers who have already covered the “Getting Started” and “Tutorial” parts of the manual, and have run at least some of the examples described there.

- The compiler itself is described in chapter 9. This includes descriptions of the language accepted by the compiler, the option switches used to operate it and a complete list of the error messages it can produce.
- The discussion of the compiler’s associated run-time library is divided into two chapters. An overview of the library, divided into sections by function, is provided in chapter 10. A detailed definition of every entry in the run-time library, arranged in alphabetical order of name, follows in chapter 11.
- The utility programs provided with the compiler are described in chapters 12 to 17.
- The configuration software takes up two chapters in this manual. First, a general description of the configuration language is given in chapter 18. This is followed by specific information about the flood-fill configurer (chapter 19).

- A number of “black box” task images are provided in the distribution disk. Task “data sheets” are provided for these in chapter 20.

## 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 in this manual is that specified by British Standard 6154[9]. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory[10].

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.

## Chapter 9

# C Compiler Reference

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.

### 9.1 The C Language

The basis of the C language adopted by 3L for Parallel C is the one given by Kernighan and Ritchie (the designers of the language) in the first edition of *The C Programming Language*[1]. The definition of C given in this book will be referred to as “K&R C”.

3L have also included in the compiler the most important features of the American national standard for the C language, as defined by ANSI X3.159-1989[3]. We shall refer to this standard dialect as “ANSI C”. ANSI C features which have been adopted by Parallel C are discussed in section 9.1.1 below.

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. For this reason, the library functions provided with Parallel C (see chapter 10) are based on the ANSI standard. These have been supplemented with functions to support “special” transputer facilities, and a number of functions which provide compatibility with older versions of the run-time library, or with run-time libraries on other systems.

In order to use Parallel C and make best use of this manual, we recommend that you should have access to the information in the first edition of *The C Programming Language*[1]. The second edition of the book[2], which describes ANSI C, may also be useful.

The differences between Parallel C and K&R C are described here.

### **9.1.1 ANSI Features**

The following ANSI C features are supported by Parallel C. The section numbers in the text refer to sections of the ANSI standard[3].

#### **9.1.1.1 Trigraph Sequences**

The ANSI trigraph sequences, as described in ANSI 2.2.1.1, are recognised by the compiler.

#### **9.1.1.2 Escape Sequences**

All the ANSI escape sequences are recognised by the compiler, including hexadecimal escape sequences of the form `\xdd`; for example, `\x0D`.

### 9.1.1.3 Translation Limits

The maximum length of a string literal is about 4 KB; see ANSI 2.2.4.1.

### 9.1.1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<code>asm</code>	<code>double</code>	<code>int</code>	<code>typedef</code>
<code>auto</code>	<code>else</code>	<code>long</code>	<code>union</code>
<code>break</code>	<code>enum</code>	<code>register</code>	<code>unsigned</code>
<code>case</code>	<code>extern</code>	<code>return</code>	<code>void</code>
<code>char</code>	<code>float</code>	<code>short</code>	<code>volatile</code>
<code>const</code>	<code>for</code>	<code>sizeof</code>	<code>while</code>
<code>continue</code>	<code>fortran</code>	<code>static</code>	
<code>default</code>	<code>goto</code>	<code>struct</code>	
<code>do</code>	<code>if</code>	<code>switch</code>	

Parallel C includes the ANSI C keywords `const`, `enum`, `void` and `volatile`. The K&R keyword `entry` is not implemented. For the uses of the `asm` keyword, see section 9.7. Although the `fortran` keyword is recognised, it currently has no function in Parallel C.

### 9.1.1.5 Identifiers

Two identifiers are deemed by the compiler to be the same if their first 255 characters match (K&R C says 8 characters). Any additional characters are ignored. For external linkage, only the first 31 characters of an identifier are significant. Case is significant, even for external identifiers processed by the linker.

The ANSI standard allows compilers to restrict the number of significant characters for external linkage to 6. For this reason, if C

programs are to be portable to many different compilers, they should only use external identifiers which are distinct in the first 6 characters whether or not the distinction between upper and lower case letters is ignored.

#### 9.1.1.6 Types

All the ANSI-specified integral types are implemented. This includes `int`, `short`, `char` and `long` with the `signed` and `unsigned` versions of each. `short` variables are 16-bit objects; `long` variables are the same size as `int` variables.

`long double` declarations are accepted; they are treated as `double`.

Enumeration (`enum`) data types are accepted.

The `void` data type is implemented.

#### 9.1.1.7 Constants

The ANSI suffixes to floating constants (`'f'`, `'l'`, `'F'`, `'L'`) and to integer constants (`'u'`, `'l'`, `'U'`, `'L'`) are accepted by the compiler, although such suffixed constants are not at present treated differently from unsuffixed ones.

Wide character constants (e.g., `L'a'`) and wide string literals (e.g., `L"hello"`) are accepted, but are treated like the corresponding non-wide elements.

Adjacent string literals are treated as a single literal. For example, the following two statements have the same effect:

```
p = "Hello, world";  
p = "Hello, ""world";
```

Note, however, that although ANSI allows white space to occur between the two literals, the compiler does not at present accept this.



### 9.1.1.8 Conversions

The compiler follows the ANSI standard by performing the “integral promotions” on integer-type values before they are used in an expression. This means that if the whole range of values of the type can be represented by an `int` it is converted into an `int`; otherwise it is converted into an `unsigned int`.

Only after this has been done are the necessary conversions for evaluating the expression performed. In particular, if at this stage one of the operands is an `unsigned int` and the other an `int`, the `int` will be converted to `unsigned int`.

This will make a difference to the value of expressions only in a small number of cases. For example,

```
unsigned char c = 5;
int a, b = -1;
a = (c > b);
```

With K&R C, `a` is assigned the value 0 (false). Parallel C follows ANSI C in assigning it the value 1 (true).

Full details of these conversions may be found in ANSI 3.2.1. Users may also be interested in the corresponding section of the *Rationale* for the standard, where this change is described as “the most serious semantic change made by the Committee to a widespread current practice”.

Note that the integral promotions treat `unsigned short` values differently for the T2 and T4/T8; for the former, they become `unsigned int`, and for the latter, `int`.

### 9.1.1.9 float expressions

Parallel C follows ANSI by performing floating-point arithmetic which involves only `float` values in single precision. This differs

from K&R C, which performs all floating-point arithmetic in double precision by default.

It is important to note that at present Parallel C does not support single-precision floating-point constants; all floating-point constants are treated as double precision. Consequently, any floating-point expression involving constants are evaluated in double precision.

If necessary, you can invoke the compiler with the `/Gd` switch, which will make it use double-precision arithmetic as required by K&R.

#### 9.1.1.10 Selecting Structure and Union Members

Parallel C follows ANSI in its treatment of the `'.'` and `"->"` operators. The first operand of the `'.'` operator must be of structure or union type, and the first operand of `"->"` must be of type pointer to structure or union. The second operand in both cases must be member of the appropriate type; Parallel C treats as an error any expression of the form `a.b` or `p.b`, where `b` does not appear in the list of members of the structure type designated by `a` or `p`.

It is possible to select the members of a structure value, such as the result of a function or the value of an assignment or conditional expression. Expressions of the form `z++->a` are also allowed.

The descriptions of the `'.'` and `"->"` operators in K&R sections 7.1 and 14.1 are both obsolete.

#### 9.1.1.11 *type name* Syntax Relaxed

K&R gives the definition of the *type name* construct (used in casts and for the `sizeof` operator) 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 ANSI C and by Parallel C.

#### 9.1.1.12 Conditional Operator

Expressions of the form `(x?a:b)` where `a` and `b` are both structures are accepted. The value of such an expression is of structure type, which may be used, for example, in an assignment (see below). The standard requires that the two structure operands are of the same type.

Conditional expressions which have second and third operands of type `void` are also allowed.

#### 9.1.1.13 Assignment to Whole struct/union Variables

Parallel C follows ANSI in allowing the assignment operator `=` to be used to assign a value to a whole `struct` variable at once. The value could be the value of another `struct` variable; or it could be a `struct` value, such as the result of a `struct` function or the value of another assignment expression or a conditional expression. The value must be of the same `struct` type as the variable to which it is being assigned. 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; } ;

void clear(struct tag item)
{
    item.p = 0;  item.q = 0;
}

int example(void)
{
    struct tag pair;

    pair.p = 3;  pair.q = 4;
    clear(pair);
    return( pair.p + pair.q );
}

```

Figure 9.1: 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 9.1 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**.

#### 9.1.1.14 Compound Assignment 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.

The anachronistic forms ‘=op’ for the assignment operators, as described in section 17 of K&R C are considered illegal. We suggest, all the same, that for portability reasons 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) */
```

#### 9.1.1.15 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 follows ANSI in making no restrictions on the use of the same member name in different structures. Programmers who wish to port their programs to other C compilers should bear this in mind.

#### 9.1.1.16 `const` and `volatile`

The ANSI type qualifier keywords `const` and `volatile` are accepted by the compiler in the appropriate contexts. However, they have no effect.

#### 9.1.1.17 Function Declarators

ANSI-style functions declarators with parameter type lists are accepted by Parallel C. These allow the compiler to check that actual and formal argument types match in function calls and definitions.

Parallel C follows ANSI by permitting functions which return structure values.

#### 9.1.1.18 Anachronistic Form of Initialisations

The anachronistic form `int x 3;` for an initialisation, which is described in section 17 of K&R C, is not allowed in Parallel C.

The correct modern form of this initialised declaration `int x 3;` would be `int x = 3;`

#### 9.1.1.19 Selection and Iteration Statements

Parallel C allows expressions of type `float` and `double` to be used as control expressions in `if` and `while` statements and as the second expression of a `for` statement.

#### 9.1.1.20 Preprocessing Directives

Preprocessor directives may be preceded on a line by white space, and the initial `#` character need not fall in the first column.

The `#error` preprocessor directive is supported. The syntax is:

```
#error text
```

A compiler error message is displayed containing the specified *text*.

The `#pragma` preprocessor directive is accepted, although there are currently no pragmas available to the user.

### 9.1.1.21 Conditional Inclusion

The `defined` unary operator, as specified in section 3.8.1 of the ANSI standard, is recognised by the preprocessor. For example:

```
#if defined(DEBUG)
    printf ("loop value=%d\n",i);
#endif
```

The `#elif` preprocessor directive is implemented.

### 9.1.1.22 Predefined Macro Names

Two of the predefined macros prescribed by ANSI are provided by Parallel C.

`__FILE__` expands to a character string literal which is the name of the current source file.

`__LINE__` expands to the line number of the current source line (a decimal constant).

## 9.1.2 Special Features

The following Parallel C features follow neither K&R C nor ANSI C.

### 9.1.2.1 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) ]; /* illegal example */
```

### 9.1.2.2 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
```

### 9.1.3 System-dependent Features

Using the features described in this section may cause different effects with different C compilers.

#### 9.1.3.1 Plain char Type Unsigned

The plain `char` data type is unsigned in Parallel C. Programs which assume that plain `chars` are signed may need to be modified.

#### 9.1.3.2 All Bit Fields Unsigned

Parallel C requires that bit fields in structures are integers. The class of integer (`int`, `short`, `long` etc.) is ignored: all bit fields are taken to be of type `unsigned int`. Bit fields specifically defined as `signed int` will be marked as errors.

This restriction is permitted by K&R C (section 8.5), although ANSI C requires that signed integer bit fields are allowed.

#### 9.1.3.3 >> 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 `7FFFFFFF16` (`LONG_MAX`) and not `FFFFFFFF16` (`-1`).

#### 9.1.3.4 Register variables

The `register` storage class is ignored in Parallel C.

## 9.2 The C main Function

The C `main` program function has the following definition.

```
#include <chan.h>
main(int argc, char *argv[], char *envp[],
      CHAN *in_ports[], int ins, CHAN *out_ports[], int outs);
```

`argc` and `argv` are described in section 3.4.2.

`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 10.5.

If your program is linked with the stand-alone library, `main` has the same arguments. However, no command-line arguments are passed to the program, and as a result `argc` is always 1, `argv[0]` is always `""`, and `argv[1]` is always `NULL`.

## 9.3 Running the Compiler

The compiler is run by one of the commands `t8c`, `t4c` or `t2c`.

`t8c` generates object code for the T800 floating-point transputer.

**t4c** generates object code for the T414 32-bit transputer.

**t2c** generates object code for the T212 16-bit transputer.

The command line used to invoke the compiler must specify a single source file name. Wild cards are not allowed.

Option switches may optionally be given on the command line. Option switches are introduced by the `'/'` character; the available switches are discussed in section 9.4 below.

If the source file is successfully compiled, a zero exit status code is returned to DOS. If errors are detected, the compiler returns an exit status code of 1. This feature can be used in DOS batch files to check whether a compilation was successful.

The compiler creates a number of temporary files as it works. Normally, these are placed in the current directory; however, the environmental variable **TMP** may be used to make the compiler put them in another directory. For example, to make the compiler place the temporary files in the root directory on disk **D:**, the following DOS command could be used.

```
C>set TMP=D:\
```

The temporary files are called **ctemp.1**, **ctemp.2** and **ctemp.3**. Usually, the compiler will delete them at the end of the run, but occasionally this may not be done; in this case, it is safe to delete them yourself.

## 9.4 Compiler Switches

This section describes the switches available to control the behaviour of the compiler. Switches are introduced by a `'/'` character 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
```

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

<i>fn</i>	An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 9.4.2 below.
<i>dir</i>	An MS-DOS filename, which will be assumed to refer to a directory.
<i>mac</i>	Any sequence of characters which is acceptable to the compiler as a macro name.
<i>str</i>	Any sequence of characters which is acceptable to the compiler as the value of a macro.
<i>n</i>	A decimal integer.

An example of a command to invoke the compiler with switches:

```
C>t8c hello /dLEVEL=3 /flkeep /w
```

This will invoke the T8 compiler to compile `hello.c`, and place a listing of the source file with any error messages in `keep.lis`. Before the compilation, a macro `LEVEL` will be defined with the value `3`. Compilation warning messages will be suppressed.

### 9.4.1 Default Switches

Switches are normally entered on the command line when the compiler is invoked. In practice, you may find you use some switches on every compilation. To avoid entering the same switches again and again, the compiler also allows switches to be entered through a DOS

environmental variable. The contents of the environmental variable `TC`, if any, are prefixed to the arguments supplied on the command line. For example, to make the compiler print its version number (`/I`) and generate debug tables (`/Zi`) every time it is run, give DOS the command: `set TC=/i/Zi`

Default options set in this way can be turned off again using the DOS command: `set TC=`

### 9.4.2 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 *fn*, but the complete MS-DOS 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 the drive specification or directory specification are omitted, then the current drive and/or directory are used.
- If a drive specification is given alone, then the output file is created in the current directory of the specified drive, regardless of the source file's directory.

The following examples may clarify this. The 'Supplied' string below is assumed to be the argument of a `/FL` switch. The current drive and directory are `c:\michael`, and the current directory on `a:` is `\output`.

Specified source file	Supplied	Output file
<code>dogs</code>	<i>nothing</i>	<code>c:\michael\dogs.lis</code>
<code>dogs</code>	<code>cats</code>	<code>c:\michael\cats.lis</code>
<code>dogs</code>	<code>cats.out</code>	<code>c:\michael\cats.out</code>
<code>dogs</code>	<code>\stuff\</code>	<code>c:\stuff\dogs.lis</code>
<code>dogs</code>	<code>a:\first\</code>	<code>a:\first\dogs.lis</code>
<code>dogs</code>	<code>a:</code>	<code>a:\output\dogs.lis</code>
<code>dogs</code>	<code>a:cats</code>	<code>a:\output\cats.lis</code>

Notice that in examples like the fourth above, it is the fact that the supplied string ends with a ‘\’ which indicates that this is a directory specification. If it is omitted, output would be sent (in this case) to `c:\stuff.lis`, even if a directory `c:\stuff` exists.

#### 9.4.2.1 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 9.4.3).

#### 9.4.2.2 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
Switches:    /T8 /FL
Compiled by: transputer C compiler, CC_transputer V2.2.2
```

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

### 9.4.3 Controlling Object Code

#### 9.4.3.1 Switches /T2, /T4, /T8 and /T8A

These switches can be used to specify which type of transputer the program is to be compiled for. /T2, /T4 and /T8 are only permitted with the `tc` command, as the `t2c`, `t4c` and `t8c` commands supply the appropriate switches automatically, and these will, in fact, appear in the “Switches:” line of the listing (see section 9.4.2.2).

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.

#### 9.4.3.2 Switch /S

By default, previous versions of the compiler followed the K&R C rule that all floating arithmetic should be carried out in double-precision. The /S switch was used to cause the compiler to perform arithmetic in single-precision floating-point when both operands of an operator were of type `float`.

The current version of the compiler by default follows the ANSI standard and uses single-precision by default. As a result, the /S switch has no effect; if you use it, a warning message will be printed and the compiler’s behaviour will be unchanged.

To cause the compiler to follow the K&R method of performing floating-point arithmetic, the `/Gd` switch should be used: see section 9.4.3.3.

#### 9.4.3.3 Switch `/Gd`

By default, the compiler now follows the ANSI standard in using single-precision floating-point arithmetic when both operands of an arithmetic operator are of type `float`.

Previous versions of the compiler, however, followed the K&R rule. *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`...”. The compiler may be made to follow the K&R rule by using the `/Gd` switch. This means that an expression like `a+b`, where `a` and `b` are `float`, is evaluated by first converting `a` and `b` to `double` and then performing the addition using double-precision floating-point arithmetic.

The new default should result in faster program execution, but because floating-point arithmetic works with approximations the numerical result of the operation may be less accurate than that obtained before. Users who are affected by this may prefer to use the `/Gd` switch.

Note that even without `/Gd`, 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`.

#### 9.4.3.4 Switch `/Gi`

By default, the compiler will when possible expand calls to certain library functions into in-line code which has the same effect. This

speeds up execution by removing the overhead of the calling instructions. The `/Gi` switch suppresses this optimisation and makes the compiler generate true function calls in these cases.

The following library functions may be expanded in-line.

<code>abs</code>	<code>chan_out_word</code>	<code>thread_priority</code>
<code>ceil</code>	<code>chan_reset</code>	<code>thread_restart</code>
<code>chan_in_byte</code>	<code>fabs</code>	<code>thread_stop</code>
<code>chan_in_message</code>	<code>floor</code>	<code>timer_after</code>
<code>chan_in_word</code>	<code>labs</code>	<code>timer_delay</code>
<code>chan_init</code>	<code>memcpy</code>	<code>timer_now</code>
<code>chan_out_byte</code>	<code>modf</code>	<code>timer_wait</code>
<code>chan_out_message</code>	<code>thread_deschedule</code>	

#### 9.4.3.5 Switch `/Gs`

Variables of type `short` and `unsigned short` are now 16 bits wide by default. If you need compatibility with earlier versions of the compiler, you can use the switch `/Gs`. This will cause the compiler to generate 32-bit `short` variables, as the earlier versions did.

Before using this switch, however, you should note that library functions, such as `printf` and `scanf`, expect `short` variables to be 16 bits wide.

A built-in macro `_3L_SHORT_BITS` has the value 16 or 32, depending on the width of `short` variables in the current compilation.

This switch has no effect on compilations for the T2, where `short` and `unsigned short` variables are always 16 bits in length.

#### 9.4.3.6 Switch `/C`

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



### 9.4.4 Controlling Code Patch Sizes

Certain constant values in a program cannot be worked out by the compiler, but must be filled in (or *patched*) by the linker. The compiler leaves gaps for these values, and fills the gaps with a special code. In some circumstances, however, the linker may decide on a patch value which is too large to fit in the gap provided by the compiler. When this happens, the linker gives the following error message:

```
FATAL ERROR(22): patch over valid code in module  module
```

The `/P` switch controls the sizes of the gaps left by the compiler, so that this situation can be avoided. There are two varieties.

#### 9.4.4.1 Switch `/PC $n$`

This switch changes the size of the gap the compiler leaves for a function call. The size of the gap limits the distance from the call to the called function. Four bits of the *displacement* are stored in every byte of gap, so the maximum displacement is  $2^{4n} - 1$  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. 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.

The compiler does not accept a `/PC1` switch, as in this case not only would the displacement be restricted to 15 bytes, but in addition backward calls would not be possible.

#### 9.4.4.2 Switch /PM $n$

A linked program contains a *module table*, which has an entry for every module in the program, including both the modules written by the user and those extracted from libraries. Each module's entry contains the address of the module's static data area. The first thing which a subprogram does is to access this address, and to do this, it must load the *module number*. These module numbers are assigned by the linker, so the compiler cannot predict how large a module's number will be. Once again, it leaves a gap, and the /PM switch allows the user to specify how large this gap is. Four bits of the module number are stored in every byte of gap, so the maximum module number is  $2^{4n} - 1$  bytes.  $n$  should be in the range 2 to 8. If the /PM switch is not used, the compiler assumes a value of 2 for  $n$ , giving a maximum module number of 255. Larger /PM numbers increase the maximum number of modules which can be linked into one program, but make the program slightly larger and slower.

If the linker reports **patch over valid code**, as described above, the likely cause is that the linked program contains more than 255 modules, including library modules. The programmer can cope with this situation as follows:

- Use /PM to increase the maximum allowable module number. For example, /PM3 will allow 4096 modules.
- Modules are assigned numbers in order, depending on their position in the linker's command line. It is essential that modules from the C library should have module numbers which are less than 255; they have already been compiled with /PM2, and this cannot be changed. So the linker command line should have the C library and harness first; then any user-written modules and libraries, compiled with a larger /PM. For example:

```
C>linkt \tc2v2\crtlt8 \tc2v2\t8harn main @mysubs,main.b4
```

### 9.4.5 Controlling Debugging

The following switches control the output of information required by the `decode` program and by Tbug, 3L's interactive symbolic debugger for the transputer.

#### 9.4.5.1 Switch /Zd

This switch causes the compiler to include line-number tables in the generated object file. These tables are used by `decode` and by Tbug to work out which piece of object code corresponds to each line of the source program. If this switch is not used, this information will not be available, and Tbug will not be able to display the source version of the program.

#### 9.4.5.2 Switch /Zi

This switch causes the compiler to include variable tables in the generated object file. These contain information about the names, locations and types of the program's identifiers. If this switch is not used, Tbug will not be able to display the variables by name and in the correct format.

The `/Zi` switch will also cause the compiler to output the line-number tables. This means that if you use `/Zi`, you do not need to use `/Zd` as well.

#### 9.4.5.3 Switch /Zo

This switch causes the compiler to generate diagnostic information in an older format which is not required for use with Tbug. This facility is retained in order to maintain compatibility with the 3L system programming environment, and is unlikely to be needed by end-users.

### 9.4.6 Controlling #include Processing

This section should be read in conjunction with section 9.6, where include file processing is discussed more fully.

#### 9.4.6.1 Switch */I**dir*

This switch adds *dir* to the include list, that is, the list of “standard places” where the compiler looks for files specified in **#include** lines. The *dir* string is assumed to be a directory, whether or not it terminates with a ‘\’.

#### 9.4.6.2 Switch */X*

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

### 9.4.7 Macro Definitions

This section should be read in conjunction with section 9.5, where predefined macros are discussed.

#### 9.4.7.1 Switch */D**mac* and */D**mac=str*

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:

```
C>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.

#### 9.4.7.2 Switch `/Umac`

This switch undefines a predefined macro—see section 9.5 for a discussion of these. This means, for example, that the following switch:

```
C>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.

### 9.4.8 Information from the Compiler

#### 9.4.8.1 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.

#### 9.4.8.2 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
Switches:    /T8 /FL /M
Compiled by: transputer C compiler, CC_transputer V2.2.2

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.

#### 9.4.8.3 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
```

#### 9.4.8.4 Switch /W

/W suppresses warning messages. The form /W*n* is allowed for compatibility with other compilers. /W0 is equivalent to /W. Other values of *n* cause warning messages to be displayed.

## 9.5 Predefined Macros

The following macros are defined with the value ‘1’ for every compilation:

```
_transputer
_3L
```

The following macros are defined to indicate which processor the current compilation is for:

Macro	Compilations defined for
_IMST2	T2C, TC/T2
_IMST4	T4C, TC/T4
_IMST8	T8C, TC/T8, T8C/T8A, TC/T8A
_IMST8A	T8C/T8A, TC/T8A

In addition, the macro \_3L\_SHORT\_BITS has the value 16 or 32, depending on the number of bits in a `short` variable for this compilation (see section 9.4.3.5 above).

Any of these predefinitions may be cancelled by the /U*mac* switch. See section 9.4.7 for details.

## 9.6 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 three ways.

- The user may define the environmental variable `3LCC_INC` to specify a series of directories, by an MS-DOS command like this:

```
C>SET 3LCC_INC=c:\root\branch;\cats
```

- If `3LCC_INC` has no value, there is only one standard place: directory `\tc2v2`. This directory is not searched if `3LCC_INC` has been defined.
- 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 9.4.6.

All the filenames which are added to the include list, either by the `SET 3LCC_INC=` command or by the `/I` compiler switch are assumed to be directories, even if they do not end with a ‘\’; in this case, the ‘\’ is supplied by the compiler. 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. Such a filename should not itself start with a ‘\’.

`\tc2v2` is the default installation directory for Parallel C. For this reason, it is also the default “standard place” to search for include files, especially header files. Its name is built into the compiler, and cannot be changed. For this reason, if you have installed the



compiler in different directory, you *must* define `3LCC_INC` to specify your installation directory. If this is not done, the compiler will be unable to find its header files.

Note also that this default directory specification does not include a disk name. This implies that by default the compiler will only search directory `\tc2v2` on the *current* disk. This problem also can be solved by defining `3LCC_INC` to include the appropriate disk name.

## 9.7 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 Inmos *Compiler Writers’ Guide* [13].

If you use assembly language you may find the `decode` utility described in chapter 14 useful. It allows you to disassemble the object files generated by the compiler and read the machine code contained in them.

### 9.7.1 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.

### 9.7.2 Assembly Language Syntax

Assembly language instructions are inserted into a program using the `asm` statement, which has the following syntax:

```
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<sup>1</sup> which take their operands from the three-register *evaluation stack*.

```
instruction =
    direct | indirect;

direct =
    direct opcode, operand;

indirect =
    indirect opcode;
```

Appendix E contains a list of the opcodes recognised by the compiler.

A function to set the transputer error flag could be written as:

```
void set_error_flag(void)
```

---

<sup>1</sup> Actually, there are only direct operations. All the indirect operations are assembled as particular literal operand values for one direct instruction called `opr`.

```
{
    asm { seterr; }
}
```

Two more example `asm` statements are shown below.

```
asm { seterr; stopp; }
asm { ldl 0; ajw -10; stl 2; ldc 123; stl 1; }
```

### 9.7.3 Literal Operands

The operand of a direct instruction can be any literal 32-bit integer value. The assembler automatically generates any `prefix` or `suffix` 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.

### 9.7.4 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);
}
```

#### 9.7.4.1 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**<sup>2</sup>. An **extern** variable can also be

---

<sup>2</sup>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.

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 */
void fun(void)
{
    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 */
void fun2(void)
{
    asm { ldc 123; stl j; }
}
```

#### 9.7.4.2 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.

### 9.7.5 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;

void sum(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;

void sum2(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;
void f1(void) { A[i] = '*'; }
```

2. Save a pointer to the object in C.

```
void f2(void) {
    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.

```
void f3(void) {
    asm {
        ldc    0x2A;    /* ASCII '*' */
        ldl    i;
        ldlp   A;
        bsub;         /* byte subscript: &A[i] */
        sb;           /* store byte */
    }
}
```

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 14 can be used to find out how the compiler has allocated storage for a program's variables.

### 9.7.6 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;

void sum3(struct s *p)
{
loop: asm {
    ld1    p;
    cj     out;
    ld1    p;
    ldnl   0;    /* p->value */
    ld1    total;
    add;
    stl    total;
    ld1    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.

#### 9.7.6.1 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. For example, the following is *incorrect*:

```

asm {
    ldc    17;
L: stl    i;    /* not allowed */
}

```

It is incorrect because a label has been put inside an `asm` statement. It must be split up as follows:

```

    asm { ldc 17; }
L: asm { stl i; }

```

#### 9.7.6.2 Jump Optimisations

The assembler always generates minimum sized jumps. Note that it may also delete unreachable jumps and merge jumps-to-jumps.

### 9.7.7 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; }
```

### 9.7.8 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 fn
```

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 section 9.9.4. 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.

## 9.8 Data-type Representations

On all transputers, a byte is 8 bits. On the T4 and the T8, a word is 32 bits (4 bytes), while on the T2, it is 16 bits (2 bytes).

### 9.8.1 Integral Data Types

On the T4 and the T8, the C integral (i.e., integer or character) data types are represented by default as follows:

Type	Bits	Bytes	Minimum	Maximum
<code>char</code>	8	1	0	255
<code>signed char</code>	8	1	-128	127
<code>short int</code>	16	2	-32768	32767
<code>unsigned short int</code>	16	2	0	65535
<code>int</code>	32	4	-2147483648	2147483647
<code>unsigned int</code>	32	4	0	4294967295
<code>long int</code>	32	4	-2147483648	2147483647

If the compiler is invoked with the `/Gs` switch, `short` variables are treated in a different way:

Type	Bits	Bytes	Minimum	Maximum
<code>short int</code>	32	4	-2147483648	2147483647
<code>unsigned short int</code>	32	4	0	65535

In this case, `unsigned short` variables occupy 32 bits, but only the 16 least significant bits are used in expressions. This interpretation allows the compiler to use the fast integer load and store operations for `unsigned shorts`, with only a simple additional masking step required when the value is to be used.

On the T2, these data types are represented as follows:

Type	Bits	Bytes	Minimum	Maximum
char	8	1	0	255
signed char	8	1	-128	127
int	16	2	-32768	32767
unsigned int	16	2	0	65535
short int	16	2	-32768	32767
unsigned short int	16	2	0	65535

The `long` data types are not supported on the T2.

### 9.8.2 Pointer Types

All pointer types (i.e., types of the form “pointer to  $x$ ”) are represented by a single word whose value is the address of the object pointed to.

### 9.8.3 Floating Types

Floating types are not supported on the T2.

The transputer follows the IEEE floating-point standard[11] in defining the representation of floating-point values. For a `float` variable, the IEEE single-precision format is used; for a `double`, the double-precision format. The way in which these standard formats are represented in transputer memory is shown in figure 9.2. Note that a `float` occupies 4 bytes (32 bits) and a `double` occupies 8 bytes (64 bits). `long double` is currently equivalent to `double`.

When the exponent field  $e$  is all ones ( $e = 255$  for single precision,  $e = 2047$  for double precision) the value represented is an *infinity* or a *Not-a-Number* (NaN). For example, the following function detects whether a given `double` value is an infinity or NaN:

```
int is_inf_or_NaN(double d)
{
    int second_word = ((int *)&d)[1];
    int e = (second_word >> 20) & 2047;
```

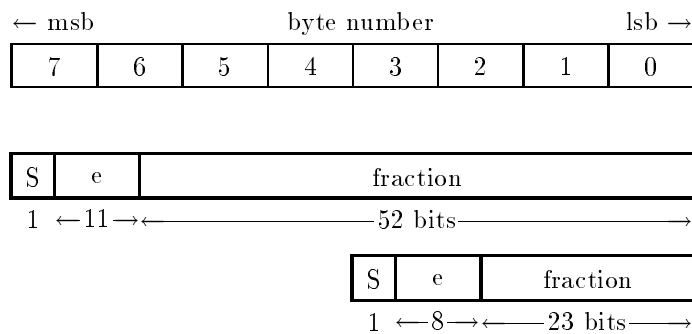


Figure 9.2: Representation of Floating-point Values

```

    return (e==2047);
}

```

An infinity is represented with a *fraction* of zero; the sign bit *s* then indicates whether positive or negative infinity is meant. A non-zero *fraction* indicates a Not-a-Number; the sign bit *s* is ignored in this case.

The following table lists all those NaNs defined by Inmos, in terms of their hexadecimal representation for both single and double precision. Note that each of these representations consists of the appropriate *e* field being set to all bits one, with the addition of a single extra bit set in the *fraction* field to indicate the type of exception.

Single	Double	Description
7FC00000	7FF80000 00000000	divide zero by zero
7FA00000	7FF40000 00000000	divide infinity by infinity
7F900000	7FF20000 00000000	multiply zero by infinity
7F880000	7FF10000 00000000	addition of opposite signed infinities
7F880000	7FF10000 00000000	subtraction of opposite signed infinities
7F840000	7FF08000 00000000	negative square root
7F820000	7FF04000 00000000	double to float conversion
7F804000	7FF00800 00000000	remainder from infinity
7F802000	7FF00400 00000000	remainder by zero
7F800010	7FF00002 00000000	result not defined mathematically
7F800008	7FF00001 00000000	result unstable
7F800004	7FF00000 80000000	result inaccurate

When an infinity or Not-a-Number is output by the standard run-time library, a special string is output instead of a normal value. See the description of `fprintf` (page 261) for details.

#### 9.8.4 Alignment and Complex Types

All types except `char` are automatically word-aligned by the compiler. Bear in mind in the discussion below that on the T4 and the T8, a *word* contains four bytes, 32 bits, whereas on the the T2 it contains two bytes, 16 bits.

`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`. On the T4 and the T8, fields may not be wider than 32 bits; on the T2, the limit is 16 bits.

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

```
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 9.3: Effect of Alignment on `struct` Size

whole number of words (even if following `char` fields could otherwise be packed into this padding space). Figure 9.3 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 1 word). They occupy the whole of this word, forcing `last` into the next word (offset 2 words), making `s1` 3 words long after being rounded up.

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. On the T4 and the T8, the overall effect is to reduce the size of the `struct` to four bytes (1 word).

## 9.9 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.

### 9.9.1 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 ‘!’.

<b>Marker</b>	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 9.9.3 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 9.4) and **error.h** (figure 9.5).

```
#include "error.h"

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

Figure 9.4: File `main.c`

```
/* error included text */

auto int count;
```

Figure 9.5: 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.h` 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.c` an opening bracket must follow the keyword `while`.

### 9.9.2 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, mis-spelt 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.

### 9.9.3 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.

### 9.9.3.1 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 decimal integer constant must not start with a 0

Because C uses a leading zero to mark the start of an octal constant, as in 01777, you cannot use leading zeros in decimal constants. The following example is therefore incorrect. It would be accepted if the 099 were replaced by 99.

```
error()
{
    int i;
    i = 099;
}
```

You would get the same error message if you accidentally typed an '8' or a '9' as part of an octal constant. The compiler would take the bad digit to mean that a decimal constant was intended, which may not start with a '0'.

- a field may not exceed 32 bits

C limits the size of a bit field to the size of an `int`. On the T4 or T8, this is 32 bits long.

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

- a field may not exceed 16 bits

C limits the size of a bit field to the size of an `int`. On the T2, this is 16 bits long.

```
error()
{
    struct thing { int huge:17; };
}
```

- 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 hexadecimal digit was expected after "\x"

The digits which follow "\x" in a hexadecimal escape sequence "\xdd" within a character or string literal must be valid hexadecimal digits: '0'-'9', 'a'-'f', or 'A'-'F'.

The example below is in error because 'G' is not a valid hexadecimal digit.

```
error()
{
    printf("\xGG", 7);
}
```

- ANSI function definition may not include parameter declarations

The cause of this message is a function definition which attempts to declare its parameters using a mixture of the K&R and ANSI notations.

```
int error(int i, float a)
int i;
float a;
{
    return a*i;
}
```

- a number is required after #line

The #line preprocessor directive keyword must be immediately followed by a decimal line number. Octal and hexadecimal forms are

not allowed. Both uses of `#line` in the example below are incorrect.

```
error()
{
  #line
  #line "72"
}
```

- 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;
{
}
```

- a parameter type declaration was expected here

In ANSI parameter list declarations, the type of each parameter must be specified explicitly. The example below would generate this error message because no type is given for `b`.

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



In C, unlike some other languages, it is not presumed that `b` must be `int` just because the previous parameter, `a`, has been declared as `int`. If `b` is meant to be `int`, you must write:

```
ok(int a, int b)
{
}
```

- *"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;};
}
```

- *"item-1" already defined as item-2 tag*

This message reports inconsistent usage of tags, for example a `struct` tag later used in a `union` declaration. *Item-1* is the name of the tag; *item-2* is `struct`, `union` or `enum` depending on how *item-1* was originally declared.

In the example below, the compiler will fault the `union` declaration, reporting that `foo` was already defined as a `struct` tag.

```
error()
{
    struct foo { int a, b; };
    union foo { int class; double x; char y; };
}
```

- 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()
{
}

```

- an identifier list in a function declarator that is not part of a function definition must be empty

A plain identifier list can only appear in a function declarator that is part of a K&R-style function definition. If the function declarator is part of an ordinary declaration, as in the example below, the brackets following the function name must be empty.

```

error()
{
    int fun(a, b, c);
    fun(1, 2, 3);
}

```

The correct way to declare `fun` here is: `int fun();`

Parameter identifiers can be used in ANSI-style function declarations anywhere, but then a type must be given for each one, as in:

```
int fun(int a, int b, int c);
```

- an object of this type cannot be initialized

An attempt has been made to initialize (in a declaration) a type of object which cannot be initialized. The example below is incorrect because it attempts to initialize a function.

```
error()
{
    int f() = 17;
}
```

- "*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;    /* struct comparison not allowed */
}
```

- 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 9.9.3.3.

- 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`. An initialized array or

structure which is declared inside a function body must be explicitly declared to be `static` or `extern`.

```
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.

DOS In the following example, the fragment `"\dir\myfile"` should be written `"\\dir\\myfile"`.

```
error()
{
    printf("cannot open \dir\myfile.dat\n");
}
```

- `&` 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, for example inside a string. 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()
{
}
```

- comment not terminated by `"*/"`

This message is given if a comment is not terminated.

```
main{}
{
    /* comment starts...
    printf ("Hello\n");
}
```

- 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; /* more than 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.

- **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/#elif/#else` without matching `#if`

The compiler control lines `#else` and `#endif` must follow a matching `#if` or `#elif` 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
}
```

- `#error:  item-1`

The `#error` preprocessor directive causes an error message of this form to be generated. *Item-1* is the message text contained in the `#error` directive. For example:

```
error()
```

```

{
  #error text of message
}

```

- error in format of integer suffix

The allowed formats for an integer suffix are laid down in section 3.1.3.2 of the ANSI standard. This error message is given if an integer suffix breaks these rules.

```

error ()
{
    int i;
    i  = 5LUL;
}

```

- 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 */
}

```

- `expected a string literal after #line n`

The `#line n` preprocessor directive may optionally be followed by a string literal representing a file name. This warning message indicates that something else was found, as in the example below. The presumed name of the source file is left unaltered.

```
warning()
{
  #line 17 foo.c
}
```

Parallel C follows ANSI in warning about an example like this one, although some other compilers will accept this older usage, with no double quotes around the file name.

- `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.

```

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 mis-spelling an identifier in the function heading or in its declaration.

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

- line break illegal in strings or character constants

This message is given when a string or character constant is not terminated on the same line as it started on. Usually this will be by mistake; however, a programmer might try to include a newline character in a string or character constant by not terminating the string. This should be done instead by using the `\n` escape sequence.

```
error()
{
    printf("line 1\nline 2");    /* correct */
    printf("line 1
line 2");                      /* wrong  */
    printf("mistake);           /* 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 mis-spelling 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 arguments does not match function prototype**

This warning message indicates that the number of actual arguments in a function call does not match the number of formal parameters in a prototype for the function which is in scope at the point of call. For example:

```
warning()
{
    void f(int);
    f(17, 99);
}
```

- **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 */
}
```

- **no support for "double" types**

This message is given for a T2 compilation if the program contains a declaration of a **double** variable or function.

```
error()
{
    double d;
}
```

- no support for "float" types

This message is given for a T2 compilation if the program contains a declaration of a `float` variable or function.

```
error()
{
    float l;
}
```

- no support for "long" types

This message is given for a T2 compilation if the program contains a declaration of a `long int` or `unsigned long int` variable or function.

```
error()
{
    long int l;
}
```

- one or more `#endif` lines inserted before extra `#else/#elif` here

This message is generated if an `#else` or `#elif` 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` or `#elif` 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 "*item-1*" must be an lvalue

The operator *item-1* requires an *lvalue* as its operand. An lvalue is an expression referring to an object in memory which can be manipulated. Note that a pointer expression is not an lvalue. To use the memory pointed to by `p` as an lvalue, you must use the expression `(*p)`, which refers to the object pointed to by `p`.

The following example is incorrect because the value of the cast is a *pointer* to the integer at address 16, not an lvalue. To use the pointed-to word and increment the integer at address 16, we would have to use the expression: `++(*(int *)16)`

```
error()
{
```

```
    ++(int *)16;  
}
```

- 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 a region of storage which can be manipulated. 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": parameter list does not match a previous prototype**

This warning message says that the (ANSI) parameter list given in the declaration of function "*item-1*" does not match the parameter list in a previous declaration of the same function. Either the parameter list has a different number of arguments from the previous declaration, or the usage of ellipsis to indicate a variable number of arguments is inconsistent between the two declarations.

For example:

```
void fun1(int i);
void fun1(int i, double d);
```

Here, the two declarations of `fun1` specify different numbers of arguments.



- *"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;
}
```

- statement out of context

A statement has been found where a declaration was expected, for example outside any function. This error may be reported if the

compiler has got out of step with the program due to previous syntax errors. If you are unsure of the cause of the error, fix any errors reported previously and recompile.

The `return` statement in the example below would be faulted, because it is not inside the body of a function.

```
return 17;

error()
{
}
```

- **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**

This error message was output by earlier versions of the compiler when it encountered a string literal which was longer than 255 characters. Now, however, the only limitation on the size of a string literal is the size of the logical line buffer. As a result, this error message should never be output.

- **struct/union/enum tag "*item-1*" not defined yet**

This message results from an attempt to declare a structure, union or enumeration variable before the tag referred to in the declaration has been declared. It is only possible to declare *pointers* to structures and unions 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 eight 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]();
}
```

- type of actual argument *item-1*, "*item-2*", does not match prototype, "*item-3*"

This warning message indicates that the type of the *item-1*th actual argument in a function call does not match the type of the corresponding formal parameter as given in a prototype for the function

which is in scope at the point of call. *Item-2* is the type of the actual argument, *item-3* is the type of the formal parameter.

In the example below, the compiler will warn of the attempt to pass a character string literal, of type `(char *)`, as an actual argument where the function prototype requires an `int`.

```
warning()
{
    void f(int);
    f("wrong");
}
```

Note that the ANSI type checking rules applied by the compiler to arguments where a function prototype is in scope are stricter than the rules applied other contexts, like assignments. In particular, pointers and integers, and different pointer types, cannot be mixed so freely.

- "*item-1*": type of parameter *item-2*, "*item-3*", does not match prototype, "*item-4*"

This warning message indicates a mismatch between the type of a parameter in the declaration of a function *item-1* and the prototype given for that parameter in some previous declaration of the same function.

*Item-2* is the number of the parameter which did not match; the leftmost parameter in the declaration is parameter one. *Item-3* is the declared parameter type in the function declaration being processed; *item-4* is the conflicting parameter type in the previous declaration of the function.

For example:

```
void fun1(int);
void fun1(double);
```

Here, the compiler will warn that in the second declaration of `fun1` the type of parameter one, "`double`", does not match the prototype, "`int`", given in the first declaration.

- type of return expression (*item-1*) incompatible with type of "*item-2*" (*item-3*)

The type of the expression in a `return` statement is incompatible with the type of the enclosing function. *Item-1* is the type of the expression, *item-2* is the identifier of the function, and *item-3* is its type. The example below is incorrect because `x` is a `struct` and cannot be returned as the result of an `int` function.

```
int error()
{
    struct { int a, b; } x;
    x.a = 0;  x.b = 17;
    return x;
}
```

- 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**

This message results from the compiler reaching the end of the source file when it was expecting more input.

```
error()
{
    int i;
    i = 5;
```

- **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 indicated section of the program in a different form.

This message was issued by previous versions of Parallel C. The current version of the compiler should not generate this message.

- **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};
}
```

- **void arguments not allowed**

In a function call, **void** actual arguments are explicitly prohibited, for example:

```
error()
{
    f( (void)0 );
}
```

### 9.9.3.2 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` switches 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 a `/`. A common cause of this error is to type, for example:

```
C>t8c cats /fo dogs.bin
```

instead of:

```
C>t8c 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 `/PCn` compilation option has been specified with an invalid value for the displacement value '*n*'. Refer to section 9.4.4.1 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 be reported to 3L 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 /T3, for example.

- target processor already specified: */switch*

This could be caused by typing either of the following:

```
C>t4/t8 cats
C>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 T4 as the target processor.

- unable to open *filename* as listing file

The compiler was unable to open the named file for output. This might be caused, for example, by using an erroneous filename:

```
C>t8c 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.

### 9.9.3.3 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.

- `*attempt to access a word at an unaligned address at line number`

This message is issued when the program attempts to load or store a word-sized object (for example, an integer) at an address which is not aligned on a word boundary. On the T2, a word contains two bytes, and this message will be given for any attempt to access a word at an odd-numbered address. On the T4 and T8, a word may only be accessed at an address which is divisible by four.

```
error()
{
    char buffer[20];
    *(int *)(buffer+7) = 200;
}
```

If this error occurs, processing of the program continues, so that further errors can be detected, but no output file is generated.

- `Error: byte initialization too complex at line number`

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-sized variable is 0 to 255 for a `char` and `-128` to `+127` for a `signed char`.

```
error()
{
    static char c = 1000;
}
```

This is treated as a fatal error.

- **Error:** shift out of range at line *number*

This message is issued when the compiler tries to fold a constant expression and discovers that the right-hand operand of a “<<” or “>>” operator is outside the allowed range, which is 0 to 32 for the T4 and T8, and 0 to 16 for the T2.

```
error()
{
    int x;
    x = 7<<50;
}
```

This is treated as a fatal error.

- **Error:** initialization too complex at line *number*

```
error()
{
    static int i;
    static int j = i;
}
```

This is treated as a fatal error.

- **Error:** Zero divide at line *number*

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;
}

```

This is treated as a fatal error.

- **Warning: integer constant truncated to 16 bits at line *number***

This message is output during a compilation for the T2, if the magnitude of an integer constant is larger than 65535.

```

error()
{
    int i;
    i = 65535;          /* no warning */
    i = -65535;         /* no warning */
    i = 65536;          /* warning */
    i = -65536;         /* warning */
}

```

This is not treated as an error; the constant is truncated, and compilation continues.

#### 9.9.4 Errors in Assembler Language

This section deals with a number of special errors which may occur in assembler language. Other errors may occur in assembler language, 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 must be redirected into a file, like this:

```
C>t8c errprog/fl > errprog.err
```

In this example, the listing, with any ordinary errors would be placed in `errprog.lis` as usual. Assembler error reports of this kind would be placed in `errprog.err`.

These error messages have the following format:

```
*opcode: message at line ln in file fn
```

*opcode* is replaced by the instruction 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 E.

```
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;  
}
```



## Chapter 10

# The C Run-Time Library

### 10.1 Introduction

#### 10.1.1 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.

The functions in the Parallel C run-time library fall into these categories.

**ANSI functions** are defined in the ANSI standard[3], chapter 4;

**Parallel functions** are required to support the special facilities of the transputer: channel communications, thread control, etc.;

**Compatibility functions** are included for compatibility purposes only, whether with earlier versions of Parallel C or with other C language environments.

ANSI functions and parallel functions are grouped by function (I/O, string handling, etc) and discussed in general in this chapter. In chapter 11 they are listed alphabetically and discussed in detail.

The compatibility functions are discussed separately in appendix F.

### 10.1.2 Versions of the Run-Time Library

The three processor types supported by Parallel C each have separate run-time libraries. The linker will detect and prohibit an attempt to link a program with the wrong version of the run-time library.

The T4 and T8 libraries each exist in two versions.

- The *Full* library contains all the functions listed in this chapter and in appendix F. To make use of the full run-time library, a program must communicate with the **afserver**, either directly or through the file-service multiplexer (see chapter 6).
- The *Stand-alone* library excludes all those functions which require the support of the **afserver**. Programs with no direct or indirect connection to the **afserver** must be linked with this variety of the run-time library. The functions it contains are marked in chapter 11 and in appendix F.

Only a stand-alone library is provided for the T2.

The stand-alone library contains only essential initialisation code, and the only I/O functions provided are those which use the transputer's channel communication facility directly; see section 10.5 below. This means that a program linked with the stand-alone library cannot use standard I/O functions like **printf**, or utility functions like **exit**.

The user-written function **main** is called by the stand-alone library with exactly the same arguments as shown in section 9.2. However, no command-line arguments are passed to the program, and as a

result `argc` is always 1, `argv[0]` is always "", and `argv[1]` is always `NULL`.

The versions of the run-time library have the following names.

	Full library	Stand-alone library
T2	<i>none</i>	<code>sacrtlt2.bin</code>
T4	<code>crtlt4.bin</code>	<code>sacrtlt4.bin</code>
T8	<code>crtlt8.bin</code>	<code>sacrtlt8.bin</code>

### 10.1.3 Conventions

This section describes how to use standard header files in calling library functions and how to interpret the notation used in chapter 11 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. Every function in the library is associated with exactly one of these header files. 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 and definitions of various related macros.

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.

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 chapter 11.

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(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`. Similarly, the synopsis for the `printf` function looks like this:

```
#include <stdio.h>
int printf(const char *format, ...);
```

The synopsis shows that `printf`'s first argument must be a character pointer, and that it is optionally followed by further arguments. The additional arguments and their allowed types are discussed in the text.

#### 10.1.4 Header Files

The following ANSI header files are supplied with the compiler. They are normally held in the compiler's installation directory (see chapter 1), which by default is `\tc2v2`.

```
assert.h  locale.h  stddef.h
ctype.h   math.h    stdio.h
errno.h   setjmp.h  stdlib.h
float.h   signal.h  string.h
limits.h  stdarg.h  time.h
```

In addition, the parallel functions of Parallel C are supported by the following header files, which are also held in the installation directory.

```
alt.h    net.h    serv.h
boot.h   par.h    thread.h
chan.h   sema.h   timer.h
dos.h
```

The following additional header files, which support certain compatibility functions, are discussed in appendix F.

```
ascii.h  chanio.h  varargs.h
```

The ANSI and parallel functions are described in the following sections which are arranged alphabetically by the names of the header files. The remainder of this section discusses the contents of four header files which for the most part contain only macro and type definitions.

### 10.1.5 Errors <errno.h>

This header contains definitions of macros which relate to the reporting of error conditions. In addition, it provides access to `errno`; users are advised not to access `errno` via a declaration of their own, as in future versions it may not simply be the identifier of an object.

### 10.1.6 Limits `<float.h>` and `<limits.h>`

These headers define a number of macros specifying the limits and characteristics of numeric types. Details may be found in section 2.2.4.2 of the ANSI standard. Note that some of these have different values depending on the processor type and the compiler's command-line switches.

### 10.1.7 Common Definitions `<stddef.h>`

This header contains definitions of the following types and macros.

<code>NULL</code>	the null pointer constant
<code>offsetof</code>	return the offset of a structure member from the start of the structure
<code>ptrdiff_t</code>	the type of the result of subtracting two pointers
<code>size_t</code>	the type of the result of <code>sizeof</code> and <code>offsetof</code>
<code>wchar_t</code>	the type of a wide character: see sections 10.18.7 and 10.18.8 below

`NULL` and `offsetof` will be discussed further in chapter 11.

## 10.2 Alt Package `<alt.h>`

The `alt` functions allow a program to input from whichever of a group of channels becomes ready first. There are two sets of functions. The `nowait` set returns a status value if none of the specified channels is ready to communicate. The others wait until a channel becomes ready. There are two ways to specify which channels are to be tested. The `_vec` functions use an array of pointers to the channels, the others use a variable-length argument list of pointers to channels.



`alt_nowait` is any one of a list of channels trying to send?

`alt_nowait_vec` is any one of an array of channels trying to send?

`alt_wait` await input from any one of a list of channels

`alt_wait_vec` await input from any one of an array of channels

### 10.3 Diagnostics <assert.h>

This header file defines the `assert` macro which assists the programmer in putting run-time diagnostics in a program.

`assert` program debugging function

### 10.4 Neighbouring Transputers <boot.h>

When a transputer is reset, and before it is booted, it executes special “peek and poke” firmware. When it is in this state, a neighbouring transputer can inspect or alter the contents of its memory by using the functions defined in this header file.

`boot_peek` peek at memory of neighbouring transputer

`boot_poke` poke into memory of neighbouring transputer

### 10.5 Channels <chan.h>

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 timeout 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, and to which channel words return after communications using them have finished. Comparing the contents of a channel word with this value provides a test for whether a thread is currently attempting to communicate over the channel.

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

## 10.6 Character Handling <ctype.h>

### 10.6.1 Character Testing Functions

The character testing 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.

<code>isalnum</code>	determines if the argument is alpha-numeric
<code>isalpha</code>	determines if the argument is alphabetic
<code>iscntrl</code>	determines if the argument is an ASCII control character
<code>isdigit</code>	determines if the argument is a digit
<code>isgraph</code>	determines if the argument is a printing character but not a space
<code>islower</code>	determines if the argument is a lowercase letter
<code>isprint</code>	determines if the argument is a printing character
<code>ispunct</code>	determines if the argument is a punctuation character
<code>isspace</code>	determines if the argument is a space, horizontal or vertical tab, carriage return, form-feed or newline
<code>isupper</code>	determines if the argument is an uppercase letter
<code>isxdigit</code>	determines if the argument is a hexadecimal digit character

### 10.6.2 Character Mapping Functions

<code>tolower</code>	converts uppercase characters to lowercase; returns lowercase characters unchanged
<code>toupper</code>	converts lowercase characters to uppercase; returns uppercase characters unchanged

## 10.7 Accessing DOS Functions `<dos.h>`

The functions described here allow a program running on a transputer system which is hosted by an MS-DOS computer to access

the software interrupts, DOS function calls and memory of the host system. The functions have been modelled after similar functions provided in native MS-DOS C compilers.

All MS-DOS functions are accessed by sending a set of register values to the host processor, executing a software interrupt instruction and finally receiving a set of modified register values. Thus, to use these functions a detailed knowledge of register uses and interrupt numbers for the MS-DOS function you wish to use is required. One source of this information is the IBM *DOS Technical Reference* [6].

The header file defines a **union** type called **REGS** which defines the register set of the host MS-DOS machine. Most of the functions described here accept two such **union** objects as arguments; one for the register values to be supplied to the interrupt routine and another to be filled with the register values after the interrupt has been called. Each **REGS** object consists of two **structs**; one **struct WORDREGS x** for the word registers (16-bit on MS-DOS machines) and another **struct BYTEREGS h** overlaying this for the equivalent byte-length registers. This overlaying arises from the fact that in the Intel 80x86 processors used to run MS-DOS, some of the 16-bit registers can also be accessed as pairs of 8-bit registers. For example, the **ax** register can be accessed as a high-order byte **ah** and a low-order byte **al**. As well as the processor registers, a **union REGS** object also contains a field representing the state of the processor **c** flag after the interrupt has been executed.

Although these **union** and **struct** data types have been closely modelled on the equivalent structures available to native MS-DOS programmers, users should note that the registers in the structures are in a different order to that conventionally used under MS-DOS, and there are some extra fields. This difference simplifies the job of the C run-time system and should not be visible to programs unless they initialise static objects of these types. Such programs would have to be changed to using the correct ordering, which can be determined from **<dos.h>**.

The 16-bit 80x86 registers such as **ax** are represented in these struc-

tures by fields declared as `unsigned short`. These will be either 16 or 32 bits wide, depending on whether the `/Gs` compiler switch is used. The functions will work correctly in both cases, and programs which access these data structures should not normally be aware of the difference.

**Host Interrupts** The functions `int86` and `int86x` call specified host software interrupts. The `...x` forms of these functions always have an extra argument which specifies the contents of the host segment registers for the call; if the non-`x` form is used, the segment registers will not be changed. The function `segread` is provided to read the contents of the segment registers so that particular registers can be changed while leaving the rest with their current values.

The functions `intdos` and `intdosx` are shorthand forms of `int86` and `int86x`; they always use host software interrupt number 21<sub>16</sub>, which is used for the main DOS function calls.

For the very simplest DOS function calls, the function `bdos` may be used; this simply takes a DOS function number and values for the `dx` and `al` registers, and causes the appropriate DOS function to be executed.

**Host Memory** Some of the more complex interrupt calls, both to MS-DOS and to add-on packages like MS-WINDOWS, require parameters and data blocks to be passed in memory rather than in registers. The Intel 80 $x$ 86 architecture uses a 32-bit quantity to specify an address in memory. The `#include` file `<dos.h>` defines a type called `pcpointer` (equivalent to `long int`) to represent these quantities. The more significant 16 bits of this object are a segment number, the least significant 16 bits are an offset from the base of that segment. These two fields can be extracted using the C shifting and masking operations. A `pcpointer` can be constructed from its components in the same way.

If a block of memory is required as a parameter to an interrupt call, it must first be acquired from MS-DOS. After use, the memory block should be returned to MS-DOS so that it may be used again. These operations can be performed either by the appropriate DOS function calls (described in the *DOS Technical Reference*) or by the run-time library functions `alloc86` and `free86`.

Because the transputer and its host MS-DOS system do not have any shared memory areas, information destined for a parameter block in the MS-DOS host cannot be simply written into the block by normal C assignment operations. Instead, a duplicate of the block is created as a C structure in the transputer's memory, and a function is then called to move the contents of the block in the transputer's memory to its counterpart in the memory of the MS-DOS host. Similarly, reading information from a block in host memory involves transferring the block into an identical structure in the transputer's memory and then accessing the latter. These two operations are performed by the run-time library functions `to86` and `from86`.

**Examples** The following program calls MS-DOS function 2<sub>16</sub> (Display Output) to display the character 'A' on the screen. The argument is passed in the dx register pair:

```
#include <dos.h>
main()
{
    bdos(0x02, /* function */
        'A',  /* dx */
        0); /* al unused */
}
```

This more complicated example uses MS-DOS function 9<sub>16</sub> (Print String) to print the string "Hello" on the screen. The string to be printed is written into a block of MS-DOS memory before the call:

```
#include <dos.h>
main()
{
    char *str = "Hello$";
```

```

union REGS r;
struct SREGS s;
pcpointer p;

/* allocate host storage and write string */
p = alloc86(strlen(str));
to86(strlen(str), str, p);

/* find current segment register values */
segread(s);

/* set up function call number */
r.h.ah = 0x09;

/* point at string to print */
r.x.dx = p & 0xffff; /* offset into ... */
s.ds = p >> 16; /* segment number */

/* perform the call */
int86x( 0x21, /* DOS function call */
        &r, /* registers in */
        &r, /* registers out */
        &s); /* segment registers */

/* free string memory in host */
free86(p);
}

```

The functions available in `dos.h` are as follows.

<code>int86</code>	perform host interrupt
<code>int86x</code>	perform host interrupt with segment registers
<code>segread</code>	read segment registers
<code>intdos</code>	perform DOS function
<code>intdosx</code>	perform DOS function with segment registers
<code>bdos</code>	perform simple BDOS function
<code>alloc86</code>	allocate host memory block



<code>free86</code>	free host memory block
<code>to86</code>	transfer memory block to host
<code>from86</code>	transfer memory block from host
<code>inp</code>	read from host I/O port
<code>outp</code>	write to host I/O port

## 10.8 Localisation <locale.h>

The localisation facility of ANSI C makes it possible to vary a number of aspects of the run-time library in order to follow local conventions regarding the format of numbers, collating sequences when comparing alphanumeric strings, the format of the time and date, and so on. Currently, Parallel C implements the "C" and "" locales only, as required by the ANSI standard.

As well as two functions, the header file defines a type, `lconv`, which contains fields relating to the formatting of numbers, and a number of macros which are used to specify aspects of the locale to change or query. For details, see section 4.4 of the standard.

<code>localeconv</code>	return details of numeric formatting conventions of the current locale
<code>setlocale</code>	change or query all or part of the locale

## 10.9 Mathematics <math.h>

The mathematical functions calculate various standard mathematical functions such as logarithms, sines, cosines etc. The header also defines the macro `HUGE_VAL` as a double expression which is returned as the result of some of the functions in certain conditions.

### 10.9.1 Treatment of Error Conditions

Errors are handled by returning impossible or unusual result values and setting an error code in the external integer variable `errno`.

### 10.9.2 Trigonometric Functions

The trigonometric functions operate on angles expressed in radians.

<code>acos</code>	returns the arc cosine of the argument
<code>asin</code>	returns the arc sine of the argument
<code>atan</code>	returns the arc tangent of the radian argument
<code>atan2</code>	returns the arc tangent of the division of the arguments
<code>cos</code>	returns the cosine of the radian argument
<code>sin</code>	returns a value that is the sine of the radian argument
<code>tan</code>	returns the tangent of the argument

### 10.9.3 Hyperbolic Functions

<code>cosh</code>	returns the hyperbolic cosine of the argument
<code>sinh</code>	returns a value that is the hyperbolic sine of the argument
<code>tanh</code>	returns the hyperbolic tangent of the argument

### 10.9.4 Exponential and Logarithmic Functions

<code>exp</code>	returns the base e raised to the power of the argument
------------------	--

<code>frexp</code>	split a floating-point number into a normalised fraction and an integral power of 2
<code>ldexp</code>	multiplies a floating-point number by an integral power of 2
<code>log</code>	returns the natural logarithm of the argument
<code>log10</code>	returns the base-ten logarithm of the argument
<code>modf</code>	breaks the argument into integral and fractional parts

### 10.9.5 Power Functions

<code>pow</code>	returns the value of the first argument raised to the power of the second argument
<code>sqrt</code>	returns the square root of the argument

### 10.9.6 Nearest Integer, Absolute Value and Remainder Functions

<code>ceil</code>	returns the smallest value which is equal to or greater than the argument
<code>fabs</code>	returns the absolute value of the floating point argument
<code>floor</code>	returns the largest integer which is less than or equal to the argument
<code>fmod</code>	calculates the floating-point remainder of the division of its arguments

## 10.10 Processor Farm Communications <net.h>

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_broadcast` send a message to every worker task

`net_send` send a message into the network

`net_receive` receive a message from the network

## 10.11 Synchronising Access to Run-Time Library <par.h>

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_fprintf` interlocked version of `fprintf`

`par_free` interlocked version of `free`

`par_malloc` interlocked version of `malloc`

`par_printf` interlocked version of `printf`

## 10.12 Semaphores <sema.h>

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_signal_n` perform `sema_signal` *n* times

`sema_test_wait` check whether waiting on a semaphore would block

`sema_wait`     perform the *wait* operation on a semaphore

`sema_wait_n`   perform `sema_wait` *n* times

## 10.13 Emulating the filter Task <serv.h>

`serv_filter`   generates Inmos file protocol filter threads

## 10.14 Nonlocal Jumps <setjmp.h>

These functions enable the programmer to save the current context of the program, and subsequently to return to it. The header defines a type `jmp_buf`, which is capable of holding all the information necessary to recreate the context.

`longjmp`       returns to the context saved by `setjmp`

`setjmp`        saves the context of the calling function for a subsequent `longjmp` call

## 10.15 Signal Handling `<signal.h>`

The signal handling package enables the programmer to create traps for various signals. In the case of Parallel C, these events do not, however, arise spontaneously, but have to be raised by the appropriate function call.

The header defines macros which are used as identifiers for the signals which can be raised, and others which define the action to be taken when a signal is raised; see the synopses in chapter 11.

<code>signal</code>	define way in which a signal is to be handled from now on
<code>raise</code>	raise a signal

## 10.16 Variable Arguments `<stdarg.h>`

A function whose declaration contains an ellipsis “...” may be called with varying numbers of arguments. The facilities described here allow such a function to access its arguments.

The header `<stdarg.h>` defines a type `va_list`. The user function should declare an object of this type, called the argument pointer, and scan the variable-length argument list with it, using these functions.

<code>va_start</code>	initialise the argument pointer
<code>va_arg</code>	find the next argument
<code>va_end</code>	finish accessing arguments

## 10.17 Input/Output <stdio.h>

The standard I/O functions provide a *portable* I/O interface for C programs. They are available in the form described here in most implementations of C. They also provide *buffering* between user programs and files or devices. This means that I/O transfers to or from real files remain efficient even if data is transferred between the file and the user program in small units (e.g., one 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

```

#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 10.1: An example of using `fopen` and `fclose`

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.

Figure 10.1 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. Section 3.4 describes the mechanism used to redefine the standard I/O streams.

### 10.17.1 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.

### 10.17.2 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, Parallel C reads and writes MS-DOS text files<sup>1</sup>. On output, newline (‘\n’) characters are converted into carriage-return line-feed sequences, and on input carriage-return line-feed sequences are converted to single newline characters. If you need to process binary data without conversion, you must inform the run-time library that a particular file is to be processed as a binary file. This can be done by using the “binary” specifier `b` in a call to `fopen` (for example, `fopen("x.bin", "rb")`.)

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.

### 10.17.3 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.

---

<sup>1</sup>This default can be changed if desired, although this procedure is not recommended. For full details, refer to the description of the `_fmode` variable on page 460.

Human-readable text files are divided into lines. Line-breaks are represented in the stream I/O model by the *newline* character, ‘\n’. 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’.

#### 10.17.4 Operations on Files

<code>remove</code>	removes a file from the file system
<code>rename</code>	renames a file
<code>tmpfile</code>	create temporary binary file
<code>tmpnam</code>	generate unique filename

#### 10.17.5 File Access Functions

<code>fclose</code>	closes a file
<code>fflush</code>	writes out any buffered information to the file
<code>fopen</code>	opens a file
<code>freopen</code>	reassigns the address of a <code>FILE</code> structure and reopens the file
<code>setbuf</code>	associates a buffer with an input or output file
<code>setvbuf</code>	determines how stream will be buffered

#### 10.17.6 Formatted Input/Output Functions

<code>fprintf</code>	performs formatted output to a specified file
<code>fscanf</code>	performs formatted input from a file

<code>printf</code>	performs formatted output to the standard output device
<code>scanf</code>	performs formatted input from the standard input device
<code>sprintf</code>	performs formatted output to a character string in memory
<code>sscanf</code>	performs formatted input from memory
<code>vfprintf</code>	similar to <code>fprintf</code> , but with a single argument instead of a list of arguments
<code>vprintf</code>	similar to <code>printf</code> , but with a single argument instead of a list of arguments
<code>vsprintf</code>	similar to <code>sprintf</code> , but with a single argument instead of a list of arguments

#### 10.17.7 Character Input/Output Functions

<code>fgetc</code>	returns the next character from a file; generates a true function call
<code>fgets</code>	reads a line from a file; the line is terminated by a NUL character
<code>fputc</code>	writes a single character to a file; generates a true function call
<code>fputs</code>	writes a string to a file
<code>getc</code>	returns the next character from a file; implemented as a macro
<code>getchar</code>	returns the next character from the standard input device

<code>gets</code>	reads a line from the standard input device; the new-line is replaced with a NUL character
<code>putc</code>	writes a single character to a file; implemented as a macro
<code>putchar</code>	writes a single character to the standard output device
<code>puts</code>	writes a string to the standard output device; terminates the string with a newline
<code>ungetc</code>	writes a character to a file buffer and leaves the file positioned before the character

### 10.17.8 Direct Input/Output Functions

<code>fread</code>	reads a specified number of items from the file
<code>fwrite</code>	writes the specified number of items to a file

### 10.17.9 File Positioning Functions

<code>fgetpos</code>	store value of file position indicator
<code>fseek</code>	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
<code>fsetpos</code>	set file position indicator
<code>ftell</code>	returns the current byte offset from the beginning of the file to the current location within the file
<code>rewind</code>	places you at the beginning of the file

### 10.17.10 Error Handling Functions

<code>clearerr</code>	resets the error and end of file indicators
<code>feof</code>	tests for end-of-file
<code>ferror</code>	returns a nonzero integer if an error occurs during read or write operations
<code>perror</code>	writes (to <code>stderr</code> ) the most recent error encountered

## 10.18 General Utilities <stdlib.h>

### 10.18.1 String Conversion Functions

<code>atof</code>	converts an ASCII string to a <code>double</code> value
<code>atoi</code>	converts an ASCII string to a <code>int</code> value
<code>atol</code>	converts an ASCII string to a <code>long</code> value
<code>strtod</code>	converts an ASCII string to a <code>double</code> value
<code>strtol</code>	converts an ASCII string to a <code>long int</code> value
<code>strtoul</code>	converts an ASCII string to an <code>unsigned long int</code> value

### 10.18.2 Pseudo-Random Sequence Generation Functions

<code>rand</code>	pseudo-random number generator
<code>srand</code>	change seed for <code>rand</code>

### 10.18.3 Memory Management Functions

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 (see section 3.5). In C, heap storage is allocated by calling a library function and remains allocated until it is explicitly released by calling **free**.

<b>calloc</b>	allocates and clears an area of memory
<b>free</b>	deallocates allocated space
<b>malloc</b>	allocates the specified number of contiguous bytes of memory
<b>realloc</b>	changes the size of an allocated area

### 10.18.4 Communication with the Environment

<b>abort</b>	abnormal program termination (unless trapped)
<b>atexit</b>	set exit handler function
<b>exit</b>	stop program
<b>getenv</b>	access environment variables
<b>system</b>	execute operating system command string

### 10.18.5 Searching and Sorting Utilities

<code>bsearch</code>	performs a binary search of an array
<code>qsort</code>	sorts an array

### 10.18.6 Integer Arithmetic Functions

<code>abs</code>	returns the absolute value of the integer argument
<code>div</code>	compute quotient and remainder of an integer division
<code>labs</code>	returns the absolute value of the <code>long int</code> argument
<code>ldiv</code>	compute quotient and remainder of a <code>long int</code> division

### 10.18.7 Multibyte Character Functions

In the present version of Parallel C, each multibyte character is one byte in length. The same applies to wide characters.

<code>mblen</code>	returns width of a multibyte character
<code>mbtowc</code>	convert a multibyte character to a wide character
<code>wctomb</code>	convert wide character to multibyte character

### 10.18.8 Multibyte String Functions

In the present version of Parallel C, multibyte strings and wide character strings both consist of a sequence of one-byte characters.

<code>mbstowcs</code>	convert multibyte string to wide character string
<code>wcstombs</code>	convert wide character string to multibyte string



## 10.19 String Handling <string.h>

The C language itself allows the manipulation of single characters. Library functions are provided to allow C programs to process variable-length strings of characters.

This header includes a definition of the macro `NULL` and of the type `size_t`.

### 10.19.1 Copying Functions

<code>memcpy</code>	copies a given number of bytes from one memory location to another; undefined for overlapping blocks
<code>memmove</code>	“safe” block move
<code>strcpy</code>	copies one string to another
<code>strncpy</code>	copies a maximum number of characters from one string to another

### 10.19.2 Concatenation Functions

<code>strcat</code>	concatenates two strings
<code>strncat</code>	concatenates two strings up to a maximum number of characters

### 10.19.3 Comparison Functions

<code>memcmp</code>	compare two blocks of memory
<code>strcmp</code>	performs lexicographic comparison of two ASCII strings
<code>strcoll</code>	compare strings using collating sequence of current locale

<code>strncmp</code>	performs lexicographic comparison of two ASCII strings (up to a maximum number of characters)
<code>strxfrm</code>	transform string using collating sequence of current locale

#### 10.19.4 Search Functions

<code>memchr</code>	locate character in block of memory
<code>strchr</code>	finds a specified character in a string
<code>strcspn</code>	returns the length of the initial part of a string which does not contain specified characters
<code>strpbrk</code>	locate first character from character set
<code>strrchr</code>	find last copy of specified character in string
<code>strspn</code>	returns the length of the initial part of a string which contains specified characters
<code>strstr</code>	locate substring within string
<code>strtok</code>	returns a pointer to the first character of a token

#### 10.19.5 Miscellaneous Functions

<code>memset</code>	overwrites each byte of an object with a given character code
<code>strerror</code>	maps <code>errno</code> codes to strings
<code>strlen</code>	returns the length of a string

## 10.20 Threads <thread.h>

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 literals `THREAD_URGENT` and `THREAD_NOTURG` to represent this.

`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

## 10.21 Date and Time <time.h>

The following functions return information about the time.

`clock` returns processor time used

`time` returns the current calendar time

Note that the ANSI functions `difftime`, `mktime`, `asctime`, `ctime`, `gmtime`, `localtime` and `strftime` are not yet implemented in Parallel C.

See also section 10.22 for functions associated with the transputer’s internal timers.

## 10.22 Transputer Timers <timer.h>

Each transputer associates a hardware timer with the group of threads executing at a particular priority. The timer associated with high-priority threads has a resolution of  $1\mu s$ , so that it ticks one million times per second. The timer for low priority threads has a resolution of  $64\mu s$  and ticks 15625 times a second. 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

## Chapter 11

# Alphabetic List of Run-time Library Entries

This chapter describes Parallel C's implementation of the ANSI C run-time library functions, as described in chapter 4 of the standard[3], and the functions supplied by 3L to support the special facilities of the transputer. 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.

This chapter does not describe functions which are included in the library only to maintain compatibility with earlier versions of Parallel C or other implementations of C. These are discussed in appendix F.

**MACRO** indicates a function which is implemented as a macro, and so may not be redefined.

**INLINE** indicates a function which is a candidate for inlining. By "inlining" is meant a technique whereby a call to the function results in the code for that function being included in the program at that point. Inlining will not be done if the compiler's `/GI` switch is used,

or if the function is used as a procedure parameter; nor will it be done, except in certain limited ways, if the appropriate header is not included.

**SA** indicates a function which is available as part of both the standard (`crtltx.bin`) and the stand-alone (`sacrtltx.bin`) libraries, and may thus be used from within a stand-alone task. Functions without the **SA** mark may only be used within tasks linked with the standard library (`crtltx.bin`).

**T2** indicates a function which is present in the T2 library, `sacrtlt2.bin`.

**DOS** indicates a function which is specific to the DOS operating system, or (when used at the start of a paragraph) indicates a paragraph of special interest to users of that operating system.

**NEW** indicates a library entry which is new with this release of Parallel C.

`NUL` is used here to indicate a character value of zero, such as used to terminate character strings. `NULL`, defined in `<stddef.h>` and several other headers, represents a generic “null pointer” value.

`_filer_handle` return server stream id of file descriptor

```
int _filer_handle(int fd, int *streamid);
```

`fd` must be a file descriptor as returned by `open`, `creat` or `fileno`. If it is not, `_filer_handle` returns 0 to indicate an error.

If `fd` is a valid file descriptor, `_filer_handle` returns a non-zero value to indicate success, and modifies the value of the `int` pointed to by `streamid` to be the `afserver` stream id on which the file is open. Note that an `afserver` “stream id” is different from the underlying DOS file handle.

`_filer_handle` is not defined in a header file.

**abort** NEW abnormal program termination

```
#include <stdlib.h>
void abort(void);
```

**abort** raises the signal **SIGABRT**. If this returns (that is, if no signal handler has been nominated for **SIGABRT** by a call to **signal**) the program is terminated, and the MS-DOS status is set to 1. Before termination, all functions registered by **atexit** will be called, and all the task's files will be closed.

**abs** INLINE SA T2 integer absolute value

```
#include <stdlib.h>
int abs(int arg);
```

**abs** returns the absolute value of its integer operand. The result returned by **abs** is not defined if **arg** is the largest negative integer.

**acos** SA calculates the arc cosine of its argument

```
#include <math.h>
double acos(double x);
```

**acos** returns the arc cosine in the range  $[0, \pi]$ . If **x** is outside the range  $[-1, +1]$ , the value **HUGE\_VAL** is returned, and **errno** is set to the value **EDOM**.

**alloc86** DOS allocate host memory

```
#include <dos.h>
pcpointer alloc86(int n);
```

This function allocates a block of at least **n** bytes in the base memory of the MS-DOS host computer and returns a pointer to it. If the

memory cannot be allocated, a null `pcpointer` is returned. The allocated memory cannot be accessed directly by the transputer program; rather, data can be moved between the transputer system and the host by means of the functions `to86` and `from86`.

Note that the Intel  $80x86$  architecture limits the amount of memory which can be contained in a single segment to  $65536$  ( $10000_{16}$ ) bytes. DOS permits allocation of more than this amount of memory using `alloc86`—always assuming that enough is free—but care must be taken in accessing locations past the first  $10000_{16}$  bytes of the allocated block. For example, the result of `alloc86(0x20000)` (allocate 128KB of host memory) might be the value (`pcpointer`)( $0x1F300000$ ). This is not a physical location in the host, but a combination of a segment value ( $1F30_{16}$ ) and an offset ( $0000_{16}$ ) from that segment. The corresponding physical address is  $1F300_{16}$ . The physical location offset 64KB from here is physical address  $2F300_{16}$ , which might be expressed with a segment value of  $2F30_{16}$  and an offset of  $0000_{16}$ , although there are other possibilities. The corresponding `pcpointer` value is  $0x2F300000$ , which is quite different from the value obtained by directly adding 64KB to the original `pcpointer` value. Thus, `pcpointer` address manipulation must always be performed by first reducing to the appropriate physical address.

`alt_nowait` SA T2 is any one of a list of channels trying to send?

```
#include <alt.h>
int alt_nowait(int n, ...);
```

Use `alt_nowait` to find out which, if any, of a set of channels is ready to communicate.

The parameter `n` is followed by a series of `CHAN *` arguments `chan0`, `chan1`, ..., which are pointers to the channels to be tested. `n` is the number of channels to be tested; it must match the actual number of channel pointers passed. For example: `alt_nowait(2, c0, c1);`



If a negative value is returned, no process was attempting to send a message on any of the channels tested.

Otherwise, the returned value will be in the range  $0 \dots n - 1$ , indicating which channel (`chan0`, `chan1`, ...) is ready to communicate. If more than one channel is simultaneously ready to communicate, one will be arbitrarily chosen.

`alt_nowait_vec` [SA] [T2] is any one of a group of channels trying to send?

```
#include <alt.h>
int alt_nowait_vec(int n, CHAN *channels[]);
```

Use `alt_nowait_vec` to find out which, if any, of a set of channels is ready to communicate.

The elements of the array `channels` are pointers to the channels to be tested. `n` is the number of elements in the array. Note that the channels themselves need not be in an array.

If a negative value is returned, no process was attempting to send a message on any of the channels tested.

Otherwise, the returned value will be in the range  $0 \dots n - 1$ ; it is then an index into the `channels` array indicating which channel is ready to communicate. If more than one channel is simultaneously ready to communicate, one will be arbitrarily chosen.

`alt_wait` [SA] [T2] await input from any of a list of channels

```
#include <alt.h>
int alt_wait(int n, ...);
```

Use `alt_wait` to block execution of the calling thread until any one of a set of channels becomes ready to communicate. No processor

time is consumed while waiting, so `alt_wait` is to be preferred over a “busy wait” loop which repeatedly calls `alt_nowait`.

The parameter `n` is followed by a series of `CHAN *` arguments `chan0`, `chan1`, ..., which are pointers to the channels. `n` is the number of channels; it must match the actual number of channel pointers passed. For example: `alt_wait(2, c0, c1);`

`alt_wait` will only return when one or more of the specified channels becomes ready to communicate. If no communication is attempted on any of these channels, it may never return.

If it does return, the returned value will be in the range  $0 \dots n-1$ , indicating which channel (`chan0`, `chan1`, ...) is ready to communicate. If more than one channel is simultaneously ready to communicate, one will be arbitrarily chosen.

`alt_wait_vec` SA T2 await input from any of a group of channels

```
#include <alt.h>
int alt_wait_vec(int n, CHAN *channels[]);
```

Use `alt_wait_vec` to block execution of the calling thread until any one of a group of channels becomes ready to communicate. No processor time is consumed while waiting, so `alt_wait_vec` is to be preferred over a “busy wait” loop which repeatedly calls `alt_nowait_vec`.

`channels` is an array of pointers to the channels. `n` is the number of elements in the array. Note that the channels themselves need not be in an array.

`alt_wait_vec` will only return when one or more of the specified channels becomes ready to communicate. If no communication is attempted on any of these channels, it may never return.

If it does return, the returned value will be in the range  $0 \dots n-1$ ; it is then an index into the `channels` array indicating which channel is

ready to communicate. If more than one channel is simultaneously ready to communicate, one will be arbitrarily chosen.

**asin** SA calculates the arc sine of its argument

```
#include <math.h>
double asin(double x);
```

**asin** returns the arc sine of its argument in the range  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ . If **x** is outside the range  $[-1, +1]$ , the value **HUGE\_VAL** is returned, and **errno** is set to the value **EDOM**.

**assert** MACRO program debugging routine

```
#include <assert.h>
void assert(int expression);
```

If the macro identifier **NDEBUG** is defined at the point in the source file where **<assert.h>** is included, use of the **assert** function will have no effect.

The **assert** function puts diagnostics into programs. The expression argument is any scalar expression. When it is executed, if **expression** is false (that is, evaluates to zero), **assert** writes a message on the standard error stream and terminates the program. The message gives the filename and line number of the **assert** call which failed.

No value is returned by **assert**.

**atan** SA arc tangent

```
#include <math.h>
double atan(double x);
```

**atan** returns the arc tangent of **x**.

**atan2** SA arc tangent of the division of its arguments

```
#include <math.h>
double atan2(double x, double y);
```

**atan2** returns the arc tangent of  $\frac{x}{y}$  in the range  $[-\pi, \pi]$ . If both arguments are zero, the value **HUGE\_VAL** is returned, and **errno** is set to the value **EDOM**.

**atexit** T2 set exit handler

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

The value of **func** is registered by the run-time library. The function it points to will be called (with no arguments) at normal program termination, when the **main** function returns or **exit** is called.

**atexit** returns 0 if **func** is registered successfully, otherwise it returns a non-zero value.

Any number of functions may be registered. The same function may be registered more than once.

**atof** SA convert string to floating point

```
#include <stdlib.h>
double atof(const char *nptr);
```

The string pointed to by **nptr** is converted to double-precision float-

ing point representation. The format accepted by `atof` is the same as that accepted by `strtod`; in fact, a call to `atof` is equivalent to

```
strtod(nptr, (char **)NULL)
```

`atoi` SA T2 convert string to integer

```
#include <stdlib.h>
int atoi(const char *nptr);
```

This function converts the string pointed to by `nptr` to integer representation. The format accepted by `atoi` is the same as that accepted by `strtol`, with a decimal base; in fact, a call to `atoi` is equivalent to

```
(int)strtol(npnt, (char **)NULL, 10)
```

`atol` SA convert string to long integer

```
#include <stdlib.h>
long atol(const char *nptr);
```

This function converts the string pointed to by `nptr` to long int representation. The format accepted by `atol` is the same as that accepted by `strtol`, with a decimal base; in fact, a call to `atol` is equivalent to

```
strtol(npnt, (char **)NULL, 10)
```

In Parallel C, `atol` is equivalent to `atoi` since `sizeof(int)` and `sizeof(long int)` are the same.

**bdos** DOS perform simple DOS function

```
#include <dos.h>
int bdos(int dosfn, int dosdx, int dosal);
```

This function performs a DOS function call interrupt on the host with the **ah** register (specifying the DOS function call number) set to **dosfn**, and with the **dx** and **al** registers set to **dosdx** and **dosal** respectively. It returns the contents of the **ax** register after DOS has processed the interrupt.

**bdos** is a shorthand form of the **int...** calls for the very simplest DOS function calls only.

**boot\_peek** SA peek in memory of neighbouring transputer

```
#include <boot.h>
int boot_peek(int ad, int val, CHAN *chan_in,
              CHAN *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 *Transputer Reference Manual*[12] and the *Compiler Writer’s Guide*[13]. 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** SA poke to memory of neighbouring transputer

```
#include <boot.h>
int boot_poke(int ad, int 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 *Transputer Reference Manual*[12] and the *Compiler Writer’s Guide*[13]. The function returns a non-zero value if the “poke” 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.

**bsearch** SA T2 binary search

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *,
                           const void *));
```

This function searches an array of objects for an element matching a given key. The result of **bsearch** is a pointer to the array element located by the search; if no match is found, a null pointer is returned.

**bsearch** is not limited to any particular data type; it is provided with a comparison function which allows it to compare two objects of the arbitrary type used by the program.

The array to be searched starts at **base** and has **nmemb** elements, each of **size** bytes. **key** points to the item to be searched for, which must have the same type as the elements of the array being searched.

The **compar** argument points to a comparison function which, given pointers to two objects of the same type as those pointed to by **key** and **base**, returns a negative integer to indicate the first is “less than” the second, a positive integer to indicate that the first object is “greater than” the second, or 0 to indicate that the two objects are “equal”.

Before calling **bsearch**, the array must be sorted into ascending order with respect to the comparison function pointed to by **compar**. This operation can often be most easily performed by the **qsort** function (see page 297) which can sort an arbitrary array into order. Like **bsearch**, it uses a comparison function to determine the ordering to be used.

**calloc** SA T2 allocates and clears an area of memory

```
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);
```

**calloc** returns a pointer to enough space for **nelem** objects of size **elsize**, or NULL if the request cannot be satisfied. The storage is initialised to zero.

**ceil** INLINE SA ceiling function

```
#include <math.h>
double ceil(double x);
```

**ceil** returns the smallest integer not less than **x**.



`chan_in_byte` INLINE SA T2 input a byte from a channel

```
#include <chan.h>
void chan_in_byte(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` SA T2 input a byte from a channel, or timeout

```
#include <chan.h>
int chan_in_byte_t(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` INLINE SA T2 initialise a channel word

```
#include <chan.h>
void chan_init(CHAN *chan);
```

This function initialises the channel word pointed to by its `chan` argument. This operation consists of writing the special value `NotProcess_P` 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` INLINE SA T2 input a message from a channel

```
#include <chan.h>
void chan_in_message(int len, char *b,
                     CHAN *chan);
```

This function reads a message of length `len` bytes from the channel pointed to by `chan` into the variable pointed to by `b`.

`chan_in_message_t` SA T2 input a message from a channel, or  
timeout

```
#include <chan.h>
int chan_in_message_t(int len, char *b, CHAN *chan,
                     int timeout);
```

This function attempts to read a message of length `len` 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` INLINE SA T2 input a word from a channel

```
#include <chan.h>
void chan_in_word(int *w, CHAN *chan);
```

This function reads a word-length message from the channel pointed to by `chan` into the integer variable pointed to by `w`. For the T4 and the T8, four bytes will be transferred; for the T2, two.

`chan_in_word_t` SA T2 input a word from a channel, or timeout

```
#include <chan.h>
int chan_in_word_t(int *w, CHAN *chan,
    int timeout);
```

This function attempts to read a word-length message from the channel pointed to by `chan` into the integer variable pointed to by `w`. For the T4 and the T8, four bytes will be expected; for the T2, two. 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` INLINE SA T2 output a byte to a channel

```
#include <chan.h>
void chan_out_byte(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` SA T2 output a byte to a channel, or timeout

```
#include <chan.h>
int chan_out_byte_t(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` INLINE SA T2 output a message to a channel

```
#include <chan.h>
void chan_out_message(int len, char *b,
                      CHAN *chan);
```

This function sends a message of length `len` bytes from the variable pointed to by `b` to the channel pointed to by `chan`.

`chan_out_message_t` SA T2 output a message to a channel, or  
timeout

```
#include <chan.h>
int chan_out_message_t(int len, char *b, CHAN *chan,
                      int timeout);
```

This function attempts to send a message of length `len` 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` INLINE SA T2 output a word to a channel

```
#include <chan.h>
void chan_out_word(int w, CHAN *chan);
```

This function sends a word-length message consisting of the value `w` to the channel pointed to by `chan`. For the T4 and the T8, four bytes will be transferred; for the T2, two.

`chan_out_word_t` SA T2 output a word to a channel, or timeout

```
#include <chan.h>
int chan_out_word_t(int w, CHAN *chan,
    int timeout);
```

This function attempts to send a word-length message consisting of the value `w` to the channel pointed to by `chan`. For the T4 and the T8, the message will be four bytes long; for the T2, two. 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` INLINE SA T2 reset a channel

```
#include <chan.h>
char *chan_reset(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` will be returned.

**clearerr** clear stream errors

```
#include <stdio.h>
void clearerr(FILE *stream);
```

**clearerr** resets any error indication on the named stream.

**clock** return processor time used

```
#include <time.h>
clock_t clock(void);
```

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>**).

**cos** SA cosine function

```
#include <math.h>
double cos(double x);
```

**cos** returns the cosine of its radian argument.

**cosh** SA hyperbolic cosine function

```
#include <math.h>
double cosh(double x);
```

**cosh** returns the hyperbolic cosine of its argument. If the magnitude of **x** is too large, **HUGE\_VAL** is returned, and **errno** is set to the value of **ERANGE**.

**div** NEW SA T2 integer division

```
#include <stdlib.h>
div_t div(int dividend, int divisor);
```

This function divides **dividend** by **divisor** and returns both the quotient and the remainder in a structure of type **div\_t**. This type is defined in **<stdlib.h>** and includes the following fields:

```
int quot;    /* contains the quotient */
int rem;     /* contains the remainder */
```

If the division is inexact, the quotient returned is the integer of lesser magnitude which is nearest to the algebraic quotient. If the result cannot be represented, the behaviour of **div** is undefined.

**errno** SA current error number

```
#include <errno.h>
int errno;
```

Some run-time library functions return a simple true/false value to indicate success or failure. For example, **fopen** (see page 260) returns a pointer to a file descriptor, or a null pointer for failure. Many library functions also set the variable **errno** to indicate the type of error in more detail. Some functions, like **sqrt** (see page 309), have only one possible error type. In the case of **sqrt**, this is **EDOM**—“domain error”—which is assigned to **errno** when the argument to **sqrt** is negative. Some other functions, such as **strtol**, may have several different distinguishable error cases; **strtol** may set **errno** either to **EDOM** or to **ERANGE**—“range error”. The different values of **errno** are defined as macros in **<errno.h>**.

The values of **errno** which a particular function uses are described along with the function. In this version of Parallel C, **errno** may also be assigned a server status code by any function which requires access to file services. For example, a failed call to **fopen** might set **errno** to 99 indicating “server operation failed”.

At entry to a program's `main` function, `errno` is zero. A run-time library function which does not detect an error does *not* guarantee to return `errno` to this initial state, although it may do so. Thus, unless `errno` is zeroed immediately before a call to a run-time library function, its value should only be examined if the call is otherwise known to have failed, by examination of the function's return value.

`exit` T2 terminate execution

```
#include <stdlib.h>
void exit(int status);
```

`exit` is the normal means of terminating program execution. It calls all the functions registered by calls to `atexit` (in reverse order of registration), and then closes all the task's files.

This call never returns.

`status` is used to tell the operating system what was the status of the terminating program. The header `<stdlib.h>` defines two macros so that this may be done in a machine-independant way. If `status` is zero or `EXIT_SUCCESS`, the program is terminating successfully. If `status` is `EXIT_FAILURE` it is terminating unsuccessfully.

DOS Under MS-DOS, the value of `status` is given to the MS-DOS result code. 0 indicates success, and 1 indicates failure.

`exp` SA  $e^x$  function

```
#include <math.h>
double exp(double x);
```

`exp` returns the exponential function of `x`.

`exp` returns `HUGE_VAL` when the correct value is too large; `errno` is set to `ERANGE`.



**fabs** INLINE SA floating absolute value

```
#include <math.h>
double fabs(double arg);
```

**fabs** returns the absolute value of **arg**.

**fclose** close a file

```
#include <stdio.h>
int fclose(FILE *stream);
```

**fclose** causes any buffers for the specified stream to be emptied, and the file to be closed. Buffers allocated by the standard I/O system are freed.

**fclose** is called automatically upon calling **exit**.

**fclose** returns non-zero if **stream** is not associated with an output file, or if buffered data cannot be transferred to that file.

**feof** MACRO is stream at end of file?

```
#include <stdio.h>
int feof(FILE *stream);
```

**feof** returns non-zero when end of file is read on the named input **stream**, otherwise zero. It is implemented as a macro, and therefore cannot be redeclared.

**ferror** MACRO tests for stream errors

```
#include <stdio.h>
int ferror(FILE *stream);
```

**ferror** returns non-zero when an error has occurred reading the named **stream**, otherwise zero. Unless cleared by **clearerr**, the er-

ror indication lasts until the stream is closed. **ferror** is implemented as a macro.

**fflush** flush stream buffer

```
#include <stdio.h>
int fflush(FILE *stream);
```

**fflush** causes any buffered data for the named output **stream** to be written to the file or device associated with that stream. The stream remains open.

**fflush** is called automatically by **close**, and when all streams are implicitly closed by **exit**.

EOF is returned if **stream** is not associated with an output file or if buffered data cannot be transferred to that file.

**fgetc** read a character from a stream

```
#include <stdio.h>
int fgetc(FILE *stream);
```

**fgetc** returns the next character from the specified input stream. Successive calls return successive characters from the stream. **fgetc** is a genuine function, unlike **getc** which is a macro.

EOF is returned at end of file or if a read error occurs.

**fgetpos** NEW store value of file position indicator

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

**fgetpos** stores the file position in the object pointed to by **pos**. (The type **fpos\_t** is defined in **stdio.h**).

If successful, **fgetpos** returns zero. If it fails, it returns a non-zero value, and sets **errno** to **EBADF** for a bad file descriptor, or **EINVAL** for any other error.

**CAREFUL!** Users should note that in the current version of Parallel C, **fgetpos** should only be used with binary files.

**fgets** read a string from a stream

```
#include <stdio.h>
char *fgets(char *str, int n, FILE *stream);
```

**fgets** reads a maximum of  $n - 1$  characters from the **stream** and stores them in the string **str**. Reading stops when a newline has been stored or when an end-of-file is encountered. The last character read into **str** is followed by a NUL character.

**fgets** normally returns **str**. If an error occurs, or if an end-of-file is encountered before any characters have been read, **fgets** returns **NULL**.

**CAREFUL!** Note that **fgets** behaves differently to **gets** (q.v.) with respect to any terminating newline character: **fgets** keeps the newline, **gets** deletes it from the string.

**floor** **INLINE** **SA** floor function

```
#include <math.h>
double floor(double x);
```

**floor** returns the largest integer not greater than **x**, expressed as a floating-point value.

**fmod** SA floating-point remainder

```
#include <math.h>
double fmod(double x, double y);
```

**fmod** returns the remainder from  $x/y$ .

**fopen** opens a file

```
#include <stdio.h>
FILE *fopen(const char *filename,
            const char *type);
```

**fopen** opens the file named by **filename** and associates a stream with it. **fopen** returns a pointer to be used to identify the stream in subsequent operations. **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 text file ('t') or a binary file ('b'). If this specifier is omitted, the file is taken to be a text file.<sup>1</sup>

The second and third parts of the **type** string may appear in any order. For example, "r+b" and "rb+" are equivalent. Some examples

---

<sup>1</sup>This default behaviour can be changed if required when porting a large application to the Parallel C environment; see the description of the `_fmode` variable (page 460) for details.

of possible values for **type** are now given, along with a description of their interpretation.

"r"	open text file for reading
"rt"	open text file for reading
"rb"	open binary file for reading
"rb+"	open binary file for update
"r+b"	open binary file for update
"w"	truncate and write to, or create, text file
"a"	append to, or create, text file
"ab"	append to, or create, binary file

**fopen** will fail if the file is to be opened for reading (**'r'**) and it does not exist. For writing (**'w'**) or appending (**'a'**), the file will be created if it does not exist.

**CAREFUL!** If a file is open to read and write (the **type** argument includes a **'+'**) it is not possible to switch directly from reading to writing or *vice versa*. Instead, there must be a call to **fseek** between them. If this is not done, the results are undefined.

**fprintf** formatted output

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format,
    ...);
```

The arguments which follow the **format** argument are output to the specified **stream**, using **putc**. The **format** argument controls the way in which the following argument list is converted for output.

The **format** argument is a character string 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 argument.

Each conversion specification is introduced by the character **'%'**. Following the **'%'** there may be the following, in the given order.

**Flags Field** This optional field includes any of the following flags, in any order:

-	The value will be left-justified.										
+	The value will always start with a sign.										
<i>space</i>	If the value does not start with a sign, a space will be placed before it.										
#	Use an “alternate form” conversion. The alternate forms depend on the conversion character, as follows: <table data-bbox="711 562 1403 856"> <tr> <td>o</td><td>Increase specified precision by one character, so that leading digit is always ‘0’.</td></tr> <tr> <td>x</td><td>Precede non-zero value by ‘0x’.</td></tr> <tr> <td>X, p</td><td>Precede non-zero value by ‘0X’.</td></tr> <tr> <td>e, E, f, g, G</td><td>Output decimal point even if no digits follow it.</td></tr> <tr> <td>g, G</td><td>Do not remove trailing zeroes.</td></tr> </table>	o	Increase specified precision by one character, so that leading digit is always ‘0’.	x	Precede non-zero value by ‘0x’.	X, p	Precede non-zero value by ‘0X’.	e, E, f, g, G	Output decimal point even if no digits follow it.	g, G	Do not remove trailing zeroes.
o	Increase specified precision by one character, so that leading digit is always ‘0’.										
x	Precede non-zero value by ‘0x’.										
X, p	Precede non-zero value by ‘0X’.										
e, E, f, g, G	Output decimal point even if no digits follow it.										
g, G	Do not remove trailing zeroes.										
0	For the d, i, o, u, x, X, p, e, E, f, g, G conversion characters, the value is padded with zeroes. If the ‘-’ flag appears as well, ‘0’ is ignored. For the d, i, o, u, x, X, p conversion characters, if a precision is specified, the ‘0’ flag is ignored.										

**Field Width** This optional field is a decimal integer; or an asterisk, in which case the value for the field width is obtained from the next argument, which should be an `int`.

The converted value is padded on the left (or on the right, if the ‘-’ flag has been specified). If a ‘0’ flag is in force, padding will be with ‘0’ characters; otherwise, it will be with spaces.

**Precision** This optional field consists of a `'.'` and either a decimal integer, or an asterisk, in which case the value for the precision is obtained from the next argument, which should be an `int`. If only a `'.'` is specified, the precision is taken as zero.

The meaning of the precision depends on the conversion character, as follows:

<code>d, i, o, u, x, X</code>	The minimum number of digits.
<code>e, E, f</code>	The number of digits to appear after the decimal point.
<code>g, G</code>	The maximum number of significant digits.
<code>s</code>	The maximum number of characters to be written from a string.

**Prefixes** This optional field may contain one of the following:

<code>h</code>	The following <code>'d', 'i', 'o', 'u', 'x'</code> or <code>'X'</code> conversion character corresponds to a <code>short int</code> or <code>unsigned short int</code> argument; or, the following <code>'n'</code> conversion character corresponds to an argument which is a pointer to a <code>short int</code> .
<code>l</code>	The following <code>'d', 'i', 'o', 'u', 'x'</code> or <code>'X'</code> conversion character corresponds to a <code>long int</code> or <code>unsigned long int</code> argument; or, the following <code>'n'</code> conversion character corresponds to an argument which is a pointer to a <code>long int</code> .
<code>L</code>	The following <code>'e', 'E', 'f', 'g'</code> or <code>'G'</code> conversion character corresponds to a <code>long double argument</code> .

**Conversion Character** The conversion characters and their meanings are:

d, i	The <b>int</b> argument is converted to decimal notation. The default precision is 1.
o, u, x, X	The <b>unsigned int</b> argument is converted to unsigned octal ('o'), unsigned decimal ('u') or unsigned hexadecimal ('x' or 'X'). The default precision is 1. When writing a hexadecimal number, the letters <b>abcdef</b> are used for x conversion, and <b>ABCDEF</b> for X conversion.
f	The <b>double</b> argument is converted to 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. The default precision is 6.
e, E	The <b>double</b> argument is converted to decimal notation of the form "[−]d.ddde[±]dd". There is one digit before the decimal point and the number after is equal to the precision specification for the argument; the default precision is 6. The 'e' conversion character generates 'e' as the exponent character, while 'E' generates 'E'.
g, G	The <b>double</b> argument is output in style 'f' or 'e'. The precision specifies the number of significant digits; the default is 1. Style 'e' is only used if the exponent after conversion is less than −4 or greater than or equal to the precision. Style 'E' is used in place of 'e' if 'G' is specified.
c	The <b>int</b> argument is converted to <b>unsigned char</b> and printed.
s	The argument is taken to be a string (character pointer) and characters from the string are printed until a NUL character is reached or until the number of characters indicated by the precision specification is reached; however, if the precision is zero or missing, all characters up to a NUL are printed.



<b>p</b>	The value of the <code>pointer-to-void</code> argument is printed as a hexadecimal number. The default precision is 8.
<b>n</b>	No output is performed. Instead, the number of characters output by this call to <code>fprintf</code> is placed in the <code>int</code> variable which the argument points at.
<b>%</b>	Print a <code>'%'</code> ; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field. The maximum length for a single converted argument is 512 characters.

`fprintf` returns the number of characters output, or a negative value if an output error occurred.

The first call on `fprintf`, `printf`, `sprintf`, `par_fprintf` or `par_printf` causes a 1KB buffer to be allocated from the heap.

**Output of Exceptional Values** If `fprintf` is asked to output a Not-a-Number or infinity value using the `'f'`, `'e'`, `'E'`, `'g'` or `'G'` conversion characters, a special string is output instead of a value in one of the normal formats. These strings are as follows:

Description	String
Positive infinity	<code>+inf</code>
Negative infinity	<code>-inf</code>
Divide zero by zero	<code>NaN:0/0</code>
Divide infinity by infinity	<code>NaN:inf/inf</code>
Multiply zero by infinity	<code>NaN:0*inf</code>
Add or subtract opposite signed infinities	<code>NaN:inf+-inf</code>
Square root of negative number	<code>NaN:-sqrt</code>
Convert NaN from <code>double</code> to <code>float</code>	<code>NaN:convNaN</code>
Remainder from infinity	<code>NaN:reminf</code>
Remainder from zero	<code>NaN:rem0</code>
Result not mathematically defined	<code>NaN:undef</code>
Result unstable	<code>NaN:unstable</code>
Result inaccurate	<code>NaN:inacc</code>
Undefined NaN	<code>NaN:???</code>

**fputc** write a character to a stream

```
#include <stdio.h>
int fputc(int cval, FILE *stream);
```

**fputc** appends the character `cval` to the specified output `stream`. It returns the character written. **fputc**, unlike **putc**, is a genuine function rather than a macro.

**fputc** returns `EOF` if an error occurs.

**fputs** write a string to a stream

```
#include <stdio.h>
int fputs(const char *str, FILE *stream);
```

**fputs** copies the NUL-terminated string `str` to the specified output `stream`. The NUL character which terminates the string is *not* written to the stream.

Note that unlike `puts`, `fputs` does not append a newline to the output string.

**fread** buffered binary input

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems,
             FILE *stream);
```

**fread** reads `nitems` objects, each of `size` bytes, from the specified input `stream` into memory at location `ptr`. It returns the number of complete items actually read. Zero is returned on error conditions or end of file.

For example, the following code fragment reads ten integer values from the file `f` into the integer array `a`:

```
#include <stdio.h>
FILE *f;
int a[10];
:
:
fread(a, sizeof(int), 10, f);
```

**free** SA T2 deallocates space obtained from the heap

```
#include <stdlib.h>
void free(void *ap);
```

**free** frees the space pointed to by `ap`, which will have been obtained originally by a call to `malloc`, `calloc` or `realloc`. If `ap` is a null pointer, no action is taken.

It is an error to attempt to free space not allocated by a call to `malloc`, `calloc` or `realloc`.

**free86** DOS free host memory

```
#include <dos.h>
void free86(pcpointer p);
```

**p** should be a **pcpointer** previously returned by **alloc86**. This function returns the block of host memory identified by **p** to MS-DOS for re-use.

**freopen** open a stream

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *type,
              FILE *stream);
```

**freopen** substitutes the named file **filename** in place of the open **stream**. It returns the original value of **stream**. The original stream is closed.

**freopen** is typically used to attach the 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 260) for a full description of the **type** string.

**freopen** returns the pointer **NULL** if **filename** cannot be accessed.

**frexp** SA split floating-point number into separate parts

```
#include <math.h>
double frexp(double value, int *exp);
```

**frexp** breaks **value** into its normalised fraction and an integral power of 2. The function returns the fractional part and the integral part is pointed to by **\*exp**.

**from86** DOS transfer memory block from host

```
#include <dos.h>
int from86(int len, pcpointer there, char *here);
```

This function transfers **len** bytes of host memory starting at **there** to a corresponding block starting at **here** in transputer memory. The function returns the number of bytes actually transferred. The host memory block used will normally have been previously allocated by a call to **alloc86**.

**fscanf** formatted input

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format,
    ...);
```

**fscanf** reads characters from the specified input **stream**, interprets them according to a **format** string and stores the results in the variables pointed to by the arguments following **format**.

The format string is regarded as a sequence of *directives*, which are processed one by one. **fscanf** tries to match each directive with characters read from the input stream; the way in which this matching is done depends on the directive. If a directive does not match with characters from the input stream, we say that a *matching error* has happened. In this case, the character which caused the error is not read, and **fscanf** returns at once. There are three types of directive:

- White space of any length will match white space of any length. If the input stream does not have white space at this point, the directive is ignored.
- A conversion specifier, which is a sequence of characters starting with a '%'. These are discussed below.
- Any other character will match the next character of the input stream if they are the same.

A conversion specifier consists of the following, in this order:

1. A character '%'.
2. A optional character '\*', indicating that the converted value is not to be stored;
3. The field width: an optional non-zero integer which specifies the maximum allowable width of the input field;
4. A prefix character, which may be one of the following:
 

h	With conversion characters 'd', 'i' and 'n', indicates that the argument points to a <b>short int</b> . With conversion characters 'o', 'u' and 'x', indicates that it points to an <b>unsigned short int</b> .
l	With conversion characters 'd', 'i' and 'n', indicates that the argument points to a <b>long int</b> . With conversion characters 'o', 'u' and 'x', indicates that it points to an <b>unsigned long int</b> . With conversion characters 'e', 'f' and 'g', indicates that it points to a <b>double</b> .
L	With conversion characters 'e', 'f' and 'g', indicates that the argument points to a <b>long double</b> .

Each conversion specifier will match a sequence of characters of a particular format, and these characters are read from the input stream. Reading stops when the first character which does not fit into this format is encountered; this character is *not* read. It is a matching error if no characters are read, that is, if not even one character would fit the assumed format for this specifier.

The character sequence which has been read is converted in one of a variety of ways, and the resulting internal value is stored in the variable pointed to by the next argument (unless a '\*' was included in the specifier). If this variable is not of an appropriate type for the value which has been converted, the effect is undefined.

The following specifiers are recognised.

- |         |   |
|---------|---|
| d       | Matches an optionally-signed decimal integer. The argument should be a pointer to an integer.   |
| i       | Matches an optionally-signed integer with a format such as would be acceptable to the <code>strtol</code> function with a <code>base</code> value of 0. This means that strings starting with “0x” or “0X” are interpreted as hexadecimal, strings starting with ‘0’ are interpreted as octal, and others as decimal. The argument should be a pointer to an integer.             |
| o       | Matches an optionally-signed octal integer. The argument should be a pointer to an integer.   |
| u       | Matches an optionally-signed decimal integer. The argument should be a pointer to an unsigned integer.  |
| x       | Matches an optionally-signed hexadecimal integer with a format such as would be acceptable to the <code>strtoul</code> function with a <code>base</code> value of 16. This means that the string may, but need not, start with “0x” or “0X”. The argument should be a pointer to an integer.  |
| e, f, g | Matches a floating-point number with a format such as would be acceptable to the <code>strtod</code> function. This means an optionally-signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an ‘E’ or ‘e’ followed by an optionally-signed integer. The argument should be a pointer to a floating-point variable. |
| s       | Matches a character string which includes no white space. The argument should be a pointer to an array of characters large enough to accept the string and a terminating NUL character, which will be added.  |

- c** Matches a sequence of characters of the length specified in the field width (1 by default). The argument should be a pointer to an array of characters large enough to accept the string. Note that unlike **'s'**, the **'c'** specifier does not skip white space; to read the next non-space character, use **"%1s"**.
- [** This specifier includes all the characters from the **'['** up to a later **']'**. The characters between the brackets are called the *scan-set*. The specifier matches a sequence of characters all of which are members of the scanset. So, for example, **"[aeiou]"** would match a sequence of vowels, of any length and in any order. The argument should be a pointer to an array of characters large enough to accept this sequence.
- If the first character of the scan-set is a **'^'**, then the specifier matches a sequence of characters *none* of which are members of the scan-set. To enable the scan-set to include a **']'**, the standard provides that if the scan-set starts with **']'** or **"^]"** this will not end the specifier and another **']'** will be needed. In other words, **"[]>]"** is a valid specifier, defining a scan-set consisting of **']'**, **')'** and **'>'**.
- p** Matches a pointer value of the format output by a **'p'** specifier in a **fprintf** function call. The argument should be a pointer to a pointer to **void**.
- n** The argument should be a pointer to an integer, and in this integer is written the number of characters read so far by this call to **fscanf**. No characters are read by the **n** specifier.
- %** Matches a **'%'** character. The complete specifier must be **"%"**. No argument is used.

The conversion specifiers **'E'**, **'G'** and **'X'** are treated as being equivalent to **'e'**, **'g'** and **'x'**. In addition, for compatibility purposes only,



'F' is accepted as being equivalent to 'lf', that is, a floating-point conversion which expects a pointer to **double** as the argument.

If an end-of-file or input error occurs before any conversion is done, **fscanf** returns **EOF**. Otherwise, it returns the number of input items successfully converted and stored. The specifier 'n' and specifiers including a '\*' do not count.

**fseek** reposition a stream

```
#include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

**fseek** sets the file position indicator of the specified stream. The new position is at the signed distance **offset** bytes from a location specified in **whence**. Three macros are provided for specifying **whence**:

**SEEK\_SET**     the start of the file

**SEEK\_CUR**     the current file position

**SEEK\_END**     the end of the file

**fseek** undoes any effects of **ungetc**.

**fseek** returns **-1** for improper seeks, or zero for normal completion.

When operating on a text file, **fseek**'s arguments are limited in the following ways:

- **offset** may only be 0.
- **whence** may only be **SEEK\_SET** or **SEEK\_END**.

The ANSI standard[3] also allows **fseek** to be applied to a text file with **whence = SEEK\_CUR** and **offset** set to a value previously obtained by applying **ftell** to the same stream. The current version of Parallel C does not support this.

**fsetpos** NEW set file position

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

**fsetpos** sets the file position of the specified stream to the position stored in the object pointed to by **pos**. This value should have been stored by an earlier call to **fgetpos**.

If successful, **fsetpos** returns zero. If it fails, it returns a non-zero value, and sets **errno** to **EBADF** for a bad file descriptor, or **EINVAL** for any other error.

CAREFUL! Users should note that in the current version of Parallel C, **fsetpos** should only be used with binary files.

**ftell** stream position enquiry

```
#include <stdio.h>
long int ftell(FILE *stream);
```

**ftell** returns the current value of the offset relative to the beginning of the file associated with the named **stream**. This offset is measured in bytes.

When operating on a text file, **ftell** may not give an accurate position unless the current position is either at the beginning or the end of the file.

**fwrite** buffered binary output

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nitems, FILE *stream);
```

**fwrite** writes **nitems** objects, each of **size** bytes, from memory at location **ptr** to the specified output **stream**. It returns the number of complete items actually written. Zero is returned on error conditions.

For example, the following code fragment writes the contents of the integer array **a** into the file **f**:

```
#include <stdio.h>
FILE *f;
int a[10];
:
:
fwrite(a, sizeof(int), 10, f);
```

**getc** MACRO read a character from a stream

```
#include <stdio.h>
int getc(FILE *stream);
```

**getc** returns the next character from the named input **stream**. Successive calls on **getc** return successive characters from the stream. **getc** is implemented as a macro.

EOF is returned on end of file or when a read error is detected.

**getchar** MACRO read a character from standard input

```
#include <stdio.h>
int getchar(void);
```

**getchar()** is identical to **getc(stdin)**. It returns the next character from the standard input stream **stdin**. **getchar** is implemented as a macro.

EOF is returned on end of file or read error conditions.

**getenv** access environment variable

```
#include <stdlib.h>
char *getenv(const char *name);
```

**name** is a pointer to a string which must be the name of an environment variable. If this environment variable is defined, **getenv** returns a pointer to the corresponding global string value; otherwise, a null pointer is returned. The **getenv** function allows a C program to access the strings placed in the MS-DOS command processor's environment by the MS-DOS **SET**, **PROMPT** and **PATH** commands. Note that, under MS-DOS, the names of all environmental variables are forced to be upper-case by the command processor. Thus, the result of the following command would be the definition of a variable called **FRED** to the value **Mixed**:

```
C>set fred=Mixed
```

If the named environment variable does not exist, **getenv** will return a null pointer. Otherwise, **getenv** will return a pointer to the value of the variable. For example, the following program fragment might print out something like **C:\COMMAND.COM**, the location of the MS-DOS command processor:

```
printf("%s\n", getenv("COMSPEC"));
```

Note that the string value pointed to by **getenv** will be valid only until the next call on **getenv**. Subsequent calls on **getenv** will overwrite the memory used for the original result. If you need to make several calls to **getenv**, you should therefore copy the value returned by **getenv** into a local string before making further calls.

**gets** read string from standard input

```
#include <stdio.h>
char *gets(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.

**CAREFUL!** Note that **gets** works differently to the similarly named **fgets** (q.v.) in its treatment of the terminating newline character: **gets** deletes the newline, **fgets** keeps it.

**inp** **DOS** read host I/O port

```
#include <dos.h>
int inp(unsigned int port);
```

The **inp** function reads a value from one of the host PC's byte input ports. The **port** argument specifies which port is to be read.

The value read is returned as the result of **inp**.

**int86** **DOS** perform host interrupt

```
#include <dos.h>
int int86(int intno, union REGS *inregs,
          union REGS *outregs);
```

This function calls host software interrupt number **intno** with the general registers set to the values in **inregs**. The register values after the interrupt has completed are placed in **outregs** and the function returns the value of the **ax** register as its result.

Note that the host processor segment registers **cs**, **ds**, **es** and **ss** are not set before the interrupt is called. If the segment registers are to be used, you should use the **int86x** function instead.

`int86x` DOS perform host interrupt with segment registers

```
#include <dos.h>
int int86x(int intno, union REGS *inregs,
           union REGS *outregs,
           struct SREGS *segregs);
```

This function calls host software interrupt number `intno` with the general registers set to the values in `inregs` and the segment registers set to the values in `segregs`. The register values after the interrupt has completed are placed in `outregs` and the function returns the value of the `ax` register as its result.

This function is useful for DOS calls which take pointers to objects, which are normally specified as the combination of a 16-bit register and a segment register. In the case where only some of the segment registers are to be modified, the function `segread` should be used first to obtain the current values of the others. Failure to do this can cause unpredictable behaviour.

`intdos` DOS perform DOS function

```
#include <dos.h>
int intdos(union REGS *inregs,
           union REGS *outregs);
```

This function calls `int86` specifying interrupt number `2116`. This is the software interrupt number by which DOS function calls are accessed. `intdos` is thus a shorthand for a common use of `int86`. Like `int86`, `intdos` returns the value of the host `ax` register after the interrupt has been processed.

`intdosx` DOS perform DOS function with segment registers

```
#include <dos.h>
int intdosx(union REGS *inregs, union REGS *outregs,
            struct SREGS *segregs);
```

This function calls `int86x` specifying interrupt number  $21_{16}$ . This is the software interrupt number by which DOS function calls are accessed. `intdosx` is thus a shorthand for a common use of `int86x`. Like `int86x`, `intdos` returns the value of the host `ax` register after the interrupt has been processed.

`isalnum` MACRO SA T2 is character alphanumeric?

```
#include <ctype.h>
int isalnum(int cval);
```

Returns  $\neq 0$  if `cval` is a letter or a digit, 0 otherwise.

`isalpha` MACRO SA T2 is character alphabetic?

```
#include <ctype.h>
int isalpha(int cval);
```

Returns  $\neq 0$  if `cval` is a letter, 0 otherwise.

`iscntrl` MACRO SA T2 ASCII control character?

```
#include <ctype.h>
int iscntrl(int cval);
```

Returns  $\neq 0$  if `cval` is an ASCII control character (code less than  $20_{16}$ , or code  $7F_{16}$ ), 0 otherwise.

`isdigit` `[MACRO]` `[SA]` `[T2]` is argument a digit?

```
#include <ctype.h>
int isdigit(int cval);
```

Returns  $\neq 0$  if `cval` is one of the digits ‘0’–‘9’, 0 otherwise.

`isgraph` `[MACRO]` `[SA]` `[T2]` printing ASCII character other than space?

```
#include <ctype.h>
int isgraph(int cval);
```

Returns  $\neq 0$  if `cval` is a printing character, codes  $21_{16}$  (!) to  $7E_{16}$  (~) inclusive. Returns 0 otherwise.

`[DOS]` Note that this function treats the character values between 128 and 255 inclusive as non-printable, although most are visible on a PC screen and on some printers.

`islower` `[MACRO]` `[SA]` `[T2]` is character lowercase?

```
#include <ctype.h>
int islower(int cval);
```

Returns  $\neq 0$  if `cval` is a lowercase letter, 0 otherwise.

`isprint` `[MACRO]` `[SA]` `[T2]` printing ASCII character?

```
#include <ctype.h>
int isprint(int cval);
```

Returns  $\neq 0$  if `cval` is a printing character, codes  $20_{16}$  (space) to  $7E_{16}$  (~) inclusive. Returns 0 otherwise.



**DOS** Note that this function treats the character values between 128 and 255 inclusive as non-printable, although most are visible on a PC screen and on some printers.

**ispunct** **MACRO** **SA** **T2** punctuation character?

```
#include <ctype.h>
int ispunct(int cval);
```

Returns  $\neq 0$  if **cval** is a punctuation character; otherwise 0. A punctuation is defined as being any printing character (see **isgraph**) which is not a letter, a digit or a space.

**isspace** **MACRO** **SA** **T2** white space character?

```
#include <ctype.h>
int isspace(int cval);
```

Returns  $\neq 0$  if **cval** is a space, horizontal or vertical tab, carriage return, newline or form feed character, 0 otherwise.

**isupper** **MACRO** **SA** **T2** is character uppercase?

```
#include <ctype.h>
int isupper(int cval);
```

Returns  $\neq 0$  if **cval** is an uppercase letter, 0 otherwise.

**isxdigit** **MACRO** **SA** **T2** printing hexadecimal digit?

```
#include <ctype.h>
int isxdigit(int cval);
```

Returns  $\neq 0$  if **cval** is a printing hexadecimal digit, 0 otherwise.

The printing hexadecimal digits are ‘0’ to ‘9’, ‘a’ to ‘f’ and ‘A’ to ‘F’.

`labs` INLINE SA `long int` absolute value

```
#include <stdlib.h>
long int labs(long int j);
```

`labs` returns the absolute value of `j`.

If `j` is the most negative `long int` value, `LONG_MIN`, the result cannot be represented and the value returned is undefined.

`ldexp` SA calculate  $x \times 2^{\text{exp}}$

```
#include <math.h>
double ldexp(double x, int exp);
```

`ldexp` returns the result of `x` multiplied by the value of two raised to the power `exp`. If the result is too large, the function returns `HUGE_VAL` and `errno` is set to the value of `ERANGE`.

`ldiv` NEW SA `long int` division

```
#include <stdlib.h>
ldiv_t ldiv(long int dividend, long int divisor);
```

This function divides `dividend` by `divisor` and returns both the quotient and the remainder in a structure of type `ldiv_t`. This type is defined in `<stdlib.h>` and includes the following fields:

```
long int quot;    /* contains the quotient */
long int rem;     /* contains the remainder */
```

If the division is inexact, the quotient returned is the integer of lesser magnitude which is nearest to the algebraic quotient. If the result cannot be represented, the behaviour of `ldiv` is undefined.

`localeconv` NEW SA T2 return numeric formatting parameters of current locale

```
#include <locale.h>
struct lconv *localeconv(void);
```

`localeconv` returns a pointer to an object of type `struct lconv`. The format of this structure is described in section 4.4 of the ANSI standard[3], and the type is defined in `locale.h`. The fields of this structure contain information about the way in which numeric values, including monetary values, are output by the run-time library with the current locale.

As the current version of Parallel C only supports locales "C" and "", as laid down by the standard, and as both of these have the same characteristics, the values returned for the various members of the `lconv` structure are always those laid down in 4.4 of the standard.

`log` SA calculates  $\log_e x$

```
#include <math.h>
double log(double x);
```

`log` returns the natural logarithm of `x`.

If `x` is negative, `log` returns `HUGE_VAL`, and `errno` is set to the value of `EDOM`. If `x` is zero, it returns `HUGE_VAL` and sets `errno` to `ERANGE`.

`log10` SA calculates  $\log_{10} x$

```
#include <math.h>
double log10(double x);
```

`log10` returns the base-ten logarithm of `x`.

If `x` is negative, `log` returns `HUGE_VAL`, and `errno` is set to the value of `EDOM`. If `x` is zero, it returns `HUGE_VAL` and sets `errno` to `ERANGE`.

`longjmp` [SA] [T2] non-local goto

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

This function, together with `setjmp`, is useful for dealing with errors encountered in a low-level subroutine of the program.

`longjmp` restores the stack environment saved in its `env` argument by an earlier call on `setjmp`. This has the effect of resuming execution immediately after that `setjmp` call.

`setjmp`'s caller can distinguish between the original return from `setjmp` and the second return caused by `longjmp` by examining `setjmp`'s return value. This is always 0 for the initial return, and the value of `longjmp`'s `val` argument for subsequent returns. If `val` is set to 0, `longjmp` will change it to a 1 in order to preserve this condition.

The function 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.

`malloc` [SA] [T2] allocates the specified number of contiguous bytes of memory

```
#include <stdlib.h>
void *malloc(size_t 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)`.

**CAREFUL!** If a request for a zero-length block is made, a pointer to a short—but real—block will be returned by `malloc`. Note, however, that programs intended to be portable to other implementations of C should not make the assumption that this is so; some other implementations return a null pointer instead.

`mblen` **NEW** **SA** **T2** return width of multi-byte character

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

If `s` is a null pointer, `mblen` returns 0, indicating that, for the current version of Parallel C, multibyte character codings are never state dependent. Otherwise, it returns the width in bytes of the multibyte character pointed to by `s`. In the current version, this will be 1, unless `s` is pointing at a null character, in which case it will be 0.

`mbstowcs` **NEW** **SA** **T2** convert multibyte string to wide character string

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s,
                size_t n);
```

The multibyte string pointed to by `s` is converted to a wide character string and stored in the array pointed to by `pwcs`. Conversion stops when a null character has been converted, or when `n` elements have been converted. `mbstowcs` returns the number of elements converted, excluding the terminating zero, if any.

Note that, in the present version of Parallel C, multibyte characters and wide characters are both one byte in length and there is no state-dependent encoding, so this function is equivalent to a string

copy. All possible element values are valid, so no error return can happen.

`mbtowc` NEW SA T2 convert multibyte character to wide character

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s,
           size_t n);
```

If `s` is a null pointer, `mbtowc` returns 0, indicating that, for the current version of Parallel C, multibyte character codings are never state dependent. Otherwise, it returns the width in bytes of the multibyte character pointed to by `s`. In the current version, this will be 1, unless `s` is pointing at a null character, in which case it will be 0.

In addition, the character pointed to by `s` will be converted to a wide character and stored in the location pointed to by `pwc`. In the current version, as both wide and multibyte characters are always 1 byte in length, this is equivalent to copying the character. The argument `n` specifies the maximum number of bytes to be scanned.

`memchr` SA T2 locate character in memory block

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

The `memchr` function searches for the value `c` (converted to an `unsigned char`) in the `n`-byte memory block starting at `s`.

The function returns a pointer to the first occurrence of `c` within the memory block. If the character is not located, a null pointer is returned.

`memcmp` SA T2 memory block compare

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
           size_t n);
```

The `memcmp` function compares the first `n` bytes of the two objects pointed to by `s1` and `s2`. The result returned will be less than, greater than, or equal to zero according to whether the object pointed to by `s1` is less than, greater than, or equal to the object pointed to by `s2`.

The comparison operation is performed one character at a time; a result will be returned when the first difference between the objects is located.

CAREFUL! When comparing complex objects, particularly when these were allocated using `malloc` from the heap, remember to take account of the following:

1. “Holes” are sometimes introduced into `struct` or `union` objects by the compiler to ensure that fields of the `struct` or `union` are correctly aligned on appropriate address boundaries. The contents of such “holes” are not defined, unless the `struct` or `union` is statically allocated, or has been explicitly initialised in its entirety by use of `memset` or `calloc`. For more detail on alignment in structures, see section 9.8.
2. Character arrays used as string variables may contain string values whose length is less than that of a previous string value held in the same array. In this case, the value may be followed by parts of the previous value, which may cause problems in a comparison using `memcmp`.

**memcpy** INLINE SA T2 memory block move

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

**memcpy** copies *n* characters from the object pointed to by **s2** into the object pointed to by **s1**. **memcpy** returns the value of **s1**.

CAREFUL! If the two objects pointed to by **s1** and **s2** overlap, the behaviour of **memcpy** is undefined. To copy from one object to another which overlaps it, or when it is not known whether the two objects overlap, you can use the **memmove** function instead of **memcpy**.

**memmove** SA T2 “safe” memory block move

```
#include <string.h>
void *memmove(void *s1, const void *s2,
              size_t n);
```

**memmove** copies *n* characters from the object pointed to by **s2** into the object pointed to by **s1**. **memmove** returns the value of **s1**.

If the two objects pointed to by **s1** and **s2** overlap, **memmove** will still perform the copy correctly. This is in contrast to **memcpy**, for which the behaviour would be undefined. If it is known that the objects pointed to by **s1** and **s2** definitely do *not* overlap, you can use the faster **memcpy** function instead of **memmove**.

**memset** SA T2 fill object with repeated byte value

```
#include <string.h>
void *memset(void *ptr, int cval, size_t num);
```

The **memset** function copies the value of **cval** (converted to an **unsigned char**) into each of the first **num** characters of the object pointed to by **ptr**.

The **memset** function returns the value of **ptr**.



**modf** INLINE SA split argument into integral and fractional parts

```
#include <math.h>
double modf(double value, double *iptr);
```

**modf** splits **value** into its integral and fractional parts. The function returns the signed fractional part and the integral part is pointed to by **\*iptr**.

**net\_broadcast** SA send a flood-filled network broadcast

```
#include <net.h>
int net_send(int nbytes, char *packet);
```

This function can be used by the master task of a flood-filled application to send a message to every worker task. It should not be used by any worker task.

The message to be sent is found starting at the location pointed to by **packet**. The **nbytes** parameter specifies the length in bytes of the message. This function is unlike **net\_receive** and **net\_send** in that the **nbytes** parameter is *not* restricted to **NET\_MAX\_PACKET\_LENGTH**. This means that the programmer does not have to split the message up into packets; this is done by **net\_broadcast**. The worker tasks receive the message by calling **net\_receive** in the usual way, possibly several times; **net\_broadcast** ensures that when the last packet is read, the **complete** parameter has the value 1 as usual.

The **net\_broadcast** function can only be used when all the worker tasks are known to be idle. Typically, this would be at the beginning of the program run, before any work packets have been sent out. Later, the master task can broadcast new data, provided a result packet has been received corresponding to every work packet sent out.

**net\_receive** [SA] receive a flood-filled network message

```
#include <net.h>
int net_receive(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 function uses the task's input port 0 to communicate with the router task.

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** [SA] send a flood-filled network message

```
#include <net.h>
int net_send(int nbytes, char *packet,
             int complete);
```

This function can be called by tasks participating in a flood-filled application to send a message into the network; the function uses the task's output port 0 to communicate with the router task. 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**. The **complete** argument should be 1, except for the case described next.

**nbytes** cannot be longer than **NET\_MAX\_PACKET\_LENGTH**, which is defined in **net.h** to be 1024. If you need to send a longer message, it must be broken up into a number of packets, each smaller than this limit. These packets must then each be sent by a separate call to **net\_send**. The last packet of such a chained message should be sent with **complete** set to 1; all the others should have **complete** set to 0. The **packet** argument should be updated for each call to point to the next part of the data to be sent. 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.

If at all possible, you should try to design your application so that chained messages are unnecessary. This is because a circuit has to be held open between the two tasks until the last packet is sent. As a result, sending long chained messages can clog up the network, blocking packets being delivered to other nodes.

If **nbytes** is less than zero or greater than **NET\_MAX\_PACKET\_LENGTH** 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.

NULL MACRO null pointer constant

NULL is defined in **stddef.h**, and also in **locale.h**, **stdio.h**, **stdlib.h** and **string.h**. It may be used as a null pointer value of any type, such as, for example, **(char \*)0** or **(int \*)0**.

`offsetof` MACRO NEW SA T2 offset of structure member

```
#include <stddef.h>
offsetof(type, member-designator);
```

This macro expands to a constant expression of type `size_t`, which has the value of the offset in bytes from the beginning of the structure to *member-designator*. The member may not be a bit-field.

`outp` DOS write to host output port

```
#include <dos.h>
void outp(unsigned int port, int byte);
```

`outp` writes the low-order byte of the integer value given as its second argument to one of the host PC's output ports. The first argument, `port`, specifies the target output port address.

`par_free` SA T2 deallocate space allocated by `par_malloc`

```
#include <par.h>
void par_free(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(FILE *stream, const char *format,
    ...);
```

`par_fprintf` provides access to the function `fprintf` in circumstances where multiple threads are active; access to the standard I/O structures in the run-time library is interlocked through the semaphore `par_sema`.

The first call on `fprintf`, `printf`, `sprintf`, `par_fprintf` or `par_printf` causes a 1KB buffer to be allocated from the heap.

`par_printf` formatted output on `stdout`

```
#include <par.h>
int par_printf(const char *format, ...);
```

`par_printf` provides access to the function `printf` in circumstances where multiple threads are active; access to the standard I/O structures in the run-time library is interlocked through the semaphore `par_sema`.

The first call on `fprintf`, `printf`, `sprintf`, `par_fprintf` or `par_printf` causes a 1KB buffer to be allocated from the heap.

`par_malloc`

SA
----

T2
----

 allocate the specified number of contiguous bytes of memory

```
#include <par.h>
char *par_malloc(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`.

**par\_sema** SA T2 semaphore for synchronising access to the run-time library

```
#include <sema.h>
SEMA par_sema;
```

When more than one thread is running, steps must be taken to ensure that only one thread at a time makes use of certain run-time library functions. A thread can ensure that this rule is not broken by waiting for the semaphore **par\_sema** before using one of these functions. After finishing with the run-time library, the thread should signal **par\_sema** so that other threads can get access.

**par\_sema** is also used by all the functions of the **par** package.

**perror** print error message

```
#include <stdio.h>
void perror(const char *s);
```

The **perror** function maps the value in the global variable **errno** into a textual message, which is printed on the standard error stream **stderr**.

If **s** is not a null pointer, **perror** first prints the string pointed to by **s** followed by a colon and a space. Regardless of the value of **s**, **perror** next prints a message corresponding to **errno** followed by a new-line character.

The error messages produced by **perror** are the same as those which can be obtained by calling the function **strerror** (see page 312) with **errno** as argument.

For example, if the current value of **errno** is **EDOM**, a call such as **perror("myprog")** might produce the following output:

```
myprog: domain error
```

**pow** SA calculates  $x^y$

```
#include <math.h>
double pow(double x, double y);
```

**pow** returns the value of **x** raised to the power of **y**.

If **x** is negative and **y** is not an integral number, **pow** returns **HUGE\_VAL** and sets **errno** to the value of **EDOM**. If **x** is zero, and **y** is zero or negative, **pow** returns **HUGE\_VAL** and sets **errno** to **EDOM**. If the result of the function is too large, **pow** returns **HUGE\_VAL** and sets **errno** to the value of **ERANGE**.

**printf** formatted output on **stdout**

```
#include <stdio.h>
int printf(const 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**. A call to **printf** is equivalent to a call to **fprintf** as follows:

```
fprintf(stdout, format, ...);
```

The first call on **fprintf**, **printf**, **sprintf**, **par\_fprintf** or **par\_printf** causes a 1KB buffer to be allocated from the heap.

**putc** MACRO writes a single character to a file

```
#include <stdio.h>
int putc(int cval, FILE *stream);
```

**putc** appends the character **cval** to the specified output **stream**. It returns the character written.

EOF is returned on error.

Because it is implemented as a macro, `putc` treats a `stream` argument with side-effects improperly. In particular, the following example causes the pointer `f` to be incremented several times, which is unlikely to be intended:

```
putc(c, *f++);
```

`putc` MACRO write a character to standard output

```
#include <stdio.h>
int putchar(int cval);
```

`putchar(cval)` is a macro defined as `putc(cval, stdout)`. The character `cval` is written to the standard output stream, `stdout` (normally the VDU).

EOF is returned on error.

`puts` write string to standard output

```
#include <stdio.h>
int puts(const char *pstr);
```

`puts` copies the NUL-terminated string `pstr` to the standard output stream `stdout` and appends a newline character. The terminating NUL character is not copied. `stdout` is normally the VDU.

`puts` appends a newline to the output string but `fputs` (q.v.) does not.



**qsort** SA T2 “quick” sort

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar) (const void *,
                          const void *));
```

This function sorts an array of items into ascending order. The array of items is pointed to by **base**; in the array, there are **nmemb** elements, with each element in the array being **size** bytes long.

Note that the type of the elements in the array is completely general: it might be **int** in a simple program or some complex **struct** type in a more sophisticated program. The definition of “ascending order” for this arbitrary data type is provided by the function **compar** which is passed to **qsort** as a parameter.

The function pointed to by **compar** takes two arguments, each a pointer to an item of the type which makes up the array pointed to by **base**. The function returns an integer less than, equal to, or greater than, zero according to whether the object pointed to by its first argument is to be regarded as less than, equal to, or greater than, that pointed to by its second argument. For example, the following function could be used as a comparison function when it is desired to sort an array of **doubles** into ascending order:

```
static int compare_doubles(double *d1, double *d2)
{
    if (*d1 < *d2) return -1; /* less */
    if (*d1 > *d2) return  1; /* greater */
    return 0;                /* else equal */
}
```

The corresponding call on **qsort** might be as follows, assuming an array **a** of 1000 **doubles**:

```
qsort(a, 1000, sizeof(double), compare_doubles);
```

Although **qsort** nominally sorts the array into ascending order, it can sort into any desired order by appropriate choice of the function passed as the **compar** argument. The array of **doubles** used in the

previous example could have been sorted into descending order of absolute value using the following comparison function:

```
#include <math.h>
static int compare_abs_doubles(double *d1, double *d2)
{
    if (fabs(*d1) < fabs(*d2)) return 1; /* less => more */
    if (fabs(*d1) > fabs(*d2)) return -1; /* more => less */
    return 0;
}
```

Here, `fabs` has been used to obtain the absolute value of the variables pointed to by each argument. The sign of the return value is opposite from that in the previous example to give the effect of reversing the order in which `qsort` will sort the array.

Once an array has been sorted into the correct order using `qsort`, the function `bsearch` (page 247) can be used to search for a particular element within the array.

**CAREFUL!** It is not usually advisable to code as follows, for example:

```
return *d1 - *d2;
```

This is because in some circumstances there could be an overflow, resulting in the items being sorted wrongly.

**raise** `NEW` `SA` `T2` raise a signal

```
#include <signal.h>
int raise(int sig);
```

This function raises the signal specified in `sig`. Macros are provided to represent the allowed values of `sig`; they are `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV` and `SIGTERM`. The action taken when the signal is raised depends on what action has been specified for that signal by a call to the `signal` function. If no such call has been made, the default action will be taken; that is, to return to the

caller's program. If such a return is made, 0 is returned if the call was successful, or 1 if there was an error.

Note that the allowed signals will only be raised in the current version by means of calls to **raise**; they will never happen spontaneously.

**rand** [SA] pseudo-random number generator

```
#include <stdlib.h>
int rand(void);
```

**rand** function returns successive pseudo-random integers in the range 0 to **RAND\_MAX**, a macro which is defined in **<stdlib.h>** to be 32767.

**realloc** [SA] [T2] changes the size of an area allocated by **malloc** or **calloc**

```
#include <stdlib.h>
void *realloc(void *ptr, size_t 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.

If **ptr** is a null pointer, the equivalent of a call to **malloc** is performed, with the specified value of **size** as the number of bytes required.

**remove** removes a file from the file system

```
#include <stdio.h>
int remove(const char *filename);
```

The **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.

Zero is returned if the file has been removed, non-zero if the operation failed.

**rename** rename a file

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

The file named **old** is renamed **new**. **old** and **new** are pointers to NUL-terminated character strings which must be valid host file names.

Zero is returned if the **rename** operation succeeds, non-zero if it fails.

The host operating system determines whether or not a particular file renaming operation will succeed.

**rewind** reposition stream to beginning

```
#include <stdio.h>
void rewind(FILE *stream);
```

**rewind(stream)** is equivalent to **(void)fseek(stream,0L,SEEK\_SET)**. 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).

**scanf** formatted input from **stdin**

```
#include <stdio.h>
int scanf(const char *format, ...);
```

**scanf** reads input from the standard input stream **stdin**. It reads characters (via **getc**), interprets them according to the given **format** and stores the resulting values in the locations pointed to by the pointer arguments following **format**.

The exact meaning of the arguments to **scanf** is the same as that of the arguments of the same name to the function **fscanf**. In fact, the call

```
scanf(format, ...);
```

is equivalent to

```
fscanf(stdin, format, ...);
```

If an end-of-file or input error occurs before any conversion is done, **fscanf** returns **EOF**. Otherwise, it returns the number of input items successfully converted and stored.

**segread** DOS read host segment registers

```
#include <dos.h>
void segread(struct SREGS *segregs);
```

This function reads the current values of the host 80x86 processor's segment registers into **segregs**.

**sema\_init** SA T2 initialise a semaphore

```
#include <sema.h>
void sema_init(SEMA *s, int v);
```

This function initialises the semaphore variable pointed to by **s** to an initial state in which:

- the queue of threads waiting for the semaphore is empty
- the value of the semaphore is  $v$ .

If a `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` [SA] [T2] perform a *signal* operation on a semaphore

```
#include <sema.h>
void sema_signal(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.

Otherwise, when there are no threads waiting for the semaphore pointed to by `s`, its value will simply 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` [SA] [T2] perform  $n$  *signal* operations on a semaphore

```
#include <sema.h>
void sema_signal_n(SEMA *s, int n);
```

This function calls the function `sema_signal`  $n$  times, in sequence.

The parameter  $n$  may be greater than or equal to zero.

`sema_test_wait` [SA] [T2] test whether waiting on a semaphore would block

```
#include <sema.h>
int sema_test_wait(SEMA *s);
```

If the semaphore pointed to by `s` has a non-zero count value, `sema_test_wait` decrements the semaphore count and returns a non-zero value.

Otherwise, the count in the semaphore is zero and a call on `sema_wait` would have blocked. In this case, `sema_test_wait` simply returns 0.

This allows a task to check to see if waiting on a semaphore would cause its execution to be suspended.

`sema_wait` [SA] [T2] perform a *wait* operation on a semaphore

```
#include <sema.h>
void sema_wait(SEMA *s);
```

If the value of the semaphore pointed to by `s` is not zero, its value is decreased by 1.

Otherwise, the value of the semaphore is 0. In this case, 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` [SA] [T2] perform  $n$  *wait* operations on a semaphore

```
#include <sema.h>
void sema_wait_n(SEMA *s, int n);
```

This function calls the function `sema_wait`  $n$  times, in sequence. The calling thread may be forced to wait at any point in the sequence.

The parameter  $n$  may be greater than or equal to zero.

`serv_filter` [SA] start Inmos file server protocol filter threads

```
#include <serv.h>
void serv_filter(CHAN *norm_in, CHAN *norm_out,
                CHAN *wide_in, CHAN *wide_out);
```

A historical problem involving first-silicon T414A transputers was solved by making the file server protocol used by the `afserver` program different to the documented protocol used by user programs. In programs which use the standard harness, the mismatch is handled by a pair of “filter” processes written in occam. In configured programs, the mismatch is usually dealt with by the purpose-built `filter` task.

This function allows a program to start a pair of threads which emulate the function of these filter processes or tasks. 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 protocol as documented by Inmos) 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 (specified by the `record32.value` protocol tag) which may be passed through the filter in either direction is 512 bytes. This restriction is the same as that imposed by the occam version of the link filter.

**setbuf** assign buffering to a stream

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

`setbuf` is used after a stream has been opened but before it is read or written. It causes the character array `buf` to be used instead of an automatically allocated buffer. If `buf` is the constant pointer `NULL`, I/O will be performed without any buffering being interposed by the `stdio` package. A macro `BUFSIZ` tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from `malloc` upon the first `getc` or `putc` on the file, except that output streams directed to the VDU and the standard error stream `stderr` are normally not buffered.

**setjmp** SA T2 save environment for longjmp

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

This function, together with **longjmp**, is useful for dealing with errors encountered in a low-level subroutine of the program.

**longjmp** restores the stack environment saved in its **env** argument by an earlier call on **setjmp**. This has the effect of resuming execution immediately after that **setjmp** call.

**setjmp**'s caller can distinguish between the original return from **setjmp** and the second return caused by **longjmp** by examining **setjmp**'s return value. This is always 0 for the initial return, and the value of **longjmp**'s **val** argument for subsequent returns. If **val** is set to 0, **longjmp** will change it to a 1 in order to preserve this condition.

The function 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.

**setlocale** NEW SA T2 query or change all or part of the locale

```
#include <locale.h>
char *setlocale(int category,
               const char *locale);
```

**setlocale** enables the user to change or query all or part of the current locale. The part of the locale to affect is specified in **category**; the following macros are provided to do this: **LC\_ALL**, **LC\_COLLATE**, **LC\_CTYPE**, **LC\_MONETARY**, **LC\_NUMERIC** and **LC\_TIME**.

If a **locale** is specified, the locale for the specified **category** will be changed to that locale, and the new locale will be returned. If **NULL** is specified for **locale**, the current value of the locale for that category will be returned. If the request cannot be honoured, **NULL** is returned.

In the current version of Parallel C, the only recognised locales are "C" and ""; these have the same characteristics, as defined in section 4.4 of the ANSI standard[3].

**setvbuf** NEW determine how stream will be buffered

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

After the specified **stream** has been opened, and before any other I/O has been performed on it, the function **setvbuf** may be used to change its buffering method to use the the specified **mode**. Macros are provided to specify the allowed modes; they are **\_IOFBF**, **\_IOLBF** and **\_IONBF**. Details may be found in section 4.9.5.6 of the ANSI standard[3].

In the current version of Parallel C, calls to **setvbuf** are not honoured, and a non-zero value is returned to indicate this.

**signal** NEW SA T2 define method of handling signal

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

**signal** defines how a specified signal will be handled from now on. The allowed values for **sig** are listed in the discussion of **raise**.

For the second argument, **func**, the programmer may specify the macros **SIG\_DFL** or **SIG\_IGN**, both of which result in the specified signal being ignored. Alternatively, the name of a function, called a *signal handler*, may be specified. In this case, when the signal is raised the signal handler is called. During the execution of the signal handler, that signal is ignored. Execution of the signal handler may be ended by calling **longjmp**, **exit** or **abort**; or by executing

a **return**, in which case execution will resume from the point where the signal was raised.

If there is an error in the call of **signal**, it will return the value of the macro **SIG\_ERR**, and set **errno** to **EINVAL**. Otherwise it will return the value of the **func** argument.

In the present version of Parallel C, signals may only be raised by calling the **raise** function. They do not happen spontaneously.

**sin** SA sine function

```
#include <math.h>
double sin(double x);
```

**sin** returns the sine of its radian argument.

**sinh** SA hyperbolic sine function

```
#include <math.h>
double sinh(double x);
```

**sinh** returns the hyperbolic sine of its argument. If the magnitude of **x** is too large, **HUGE\_VAL** is returned, and **errno** is set to the value of **ERANGE**.

**sprintf** SA formatted output to a string

```
#include <stdio.h>
int sprintf(char *pstr, const char *format, ...);
```

**sprintf** writes formatted output into a character array via a pointer **pstr** supplied by the caller. It returns the number of characters written into the array.

The meaning of **format** string and the use of the other arguments 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**.

The first call on **fprintf**, **printf**, **sprintf**, **par\_fprintf** or **par\_printf** causes a 1KB buffer to be allocated from the heap.

**sqrt** SA calculates  $\sqrt{x}$

```
#include <math.h>
double sqrt(double x);
```

**sqrt** returns the square root of **x**.

**sqrt** returns HUGE\_VAL when **x** is negative; **errno** is set to EDOM.

**srand** SA new seed for rand function

```
#include <stdlib.h>
void srand(unsigned int seed);
```

The **srand** function uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**.

**sscanf** formatted input from string

```
#include <stdio.h>
int sscanf(char *pstr, const char *format, ...);
```

**sscanf** reads input from the string **pstr**. It interprets the characters it reads according to the given **format** string and stores the resulting values in the locations pointed to by the pointer arguments following **format**.

The exact meaning of the arguments to **sscanf** is the same as for **fscanf**.

If the end of the string is found before any conversion is done, **fscanf** returns **EOF**. Otherwise, it returns the number of input items successfully converted and stored.

**strcat** SA T2 concatenates two strings

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

**strcat** appends a copy of string **s2** to the end of string **s1**. A pointer to the NUL-terminated result is returned.

**strchr** SA T2 find a specified character in a string

```
#include <string.h>
char *strchr(const char *pstr, int cval);
```

**strchr** locates the first occurrence of **cval** (converted to a **char**) in the string pointed to by **pstr**. The terminating NUL character is considered to be part of the string. The function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

**strcmp** SA T2 string compare

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

**strcmp** compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether **s1** is lexicographically greater than, equal to or less than **s2**.

**strcoll** SA T2 string compare using current locale's collating sequence

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

**strcmp** compares its arguments, interpreting both in the light of the **LC\_COLLATE** category of the current locale. It then returns an integer greater than, equal to, or less than 0, depending on whether **s1** is lexicographically greater than, equal to or less than **s2**.

Note that as the current version of Parallel C only supports the "C" and "" locales, **strcoll** is equivalent to a call on **strcmp**.

**strcpy** SA T2 string copy

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

**strcpy** copies string **s2** to **s1**, stopping after the NUL character has been moved. **s1** is returned. If copying takes place between objects that overlap, the behaviour is undefined.

**strcspn** SA T2 find length of string that does not contain specified characters

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

**strcspn** calculates the length of the initial part of the string pointed to by **s1** which consists of characters not from the string pointed to by **s2**. The terminating NUL character is not considered part of **s2**. The function returns the length of the part.

**strerror** map error number to message

```
#include <string.h>
char *strerror(int errnum);
```

This function maps the error number in **errnum** into a textual error message string, to which it returns a pointer. For example, an **errnum** argument of **EDOM** might return a pointer to the string "domain error".

**CAREFUL!** The string whose address is returned by **strerror** must not be modified by the caller of **strerror**. In addition, subsequent calls to **strerror** may overwrite this string with a new error message. Thus, if the result of **strerror** is not to be used immediately (for example, to be printed out) it should be copied elsewhere until it is needed to avoid being overwritten.

**strlen** SA T2 string length

```
#include <string.h>
size_t strlen(const char *pstr);
```

**strlen** returns the number of non-NUL characters in **pstr**.

**strncat** SA T2 string concatenate

```
#include <string.h>
char *strncat(char *s1, const char *s2,
              size_t num);
```

**strncat** appends a copy of string **s2** to the end of string **s1**. It copies at most **num** characters. A pointer to the NUL-terminated result is returned.



**strncmp** SA T2 string compare

```
#include <string.h>
int strncmp(const char *s1, const char *s2,
            size_t num);
```

**strncmp** compares its arguments and returns an integer greater than, equal to, or less than 0, depending on whether **s1** is lexicographically greater than, equal to or less than **s2**. At most **num** characters are looked at.

**strncpy** SA T2 string copy

```
#include <string.h>
char *strncpy(char *s1, const char *s2,
              size_t 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.

**strpbrk** SA T2 locate first character from character set

```
#include <string.h>
char *strpbrk(const char *str, const char *cset);
```

The **strpbrk** function scans the string pointed to by **str** for the first character in that string which is also contained in the string pointed to by **cset**. It returns a pointer to this character once located. If the string pointed to by **str** does not contain any of the characters from the string pointed to by **cset** then **strpbrk** returns a null pointer.

The following example shows how **strpbrk** might be used to scan a string, replacing any vowels with the character ‘\*’:

```
char str[] = "this is some example text";
char *p;
```

```
while (p = strpbrk(str, "aeiouAEIOU"))
    *p = '*';
```

After execution of this code fragment, the array `str` would contain the string `"th*s *s s*me *x*mpl* t*xt"`.

**strrchr** SA T2 find last copy of specified character in string

```
#include <string.h>
char *strrchr(char *s, int c);
```

This function locates the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. It returns a pointer to the located copy of `c`. If no copy of `c` can be located in the string, a null pointer is returned.

Note that `strrchr` treats the NUL character which terminates the string pointed to by `s` to be part of that string; therefore, a call such as `strrchr(s,0)` will locate that NUL terminator.

**strspn** SA T2 find length of string which contains specified characters

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

`strspn` calculates the length of the initial part of the string pointed to by `s1` which consists of characters from the string pointed to by `s2`. The function returns the length of the segment.

**strstr** SA T2 locate substring within string

```
#include <string.h>
char *strstr(const char *str, const char *sub);
```

This function searches for the string pointed to by **sub** within the string pointed to by **str**. If the substring cannot be located, a null pointer is returned. Otherwise, **strstr** returns a pointer to the first occurrence of the substring.

If **sub** points to an empty string (i.e., just to a NUL character) then **strstr** returns **str**.

As an example of the use of **strstr**, consider the following code fragment:

```
char *str = "The quick fox jumps.";
char *sub1 = "fox";
char *sub2 = "dog";
char *ans1 = strstr(str, sub1);
char *ans2 = strstr(str, sub2);
```

After the execution of this code fragment, **ans1** will contain a pointer to the part of **str** starting at “fox”, i.e., “fox jumps.”. On the other hand, **str** does not contain the substring “dog”, so **ans2** will contain a null pointer.

**strtod** NEW SA convert string to double value

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Starting from the place pointed to by **nptr**, **strtod** skips over initial white space, then attempts to interpret characters as forming part of a floating-point constant. Conversion stops at the first character which does not fit into the format of the constant.

The format expected is: an optional sign, a sequence of digits optionally including a decimal point, then an optional exponent part,

consisting of an ‘e’ or ‘E’, followed by an optionally-signed integer. The value of this constant is returned as the value of the function, and the object pointed to by `endptr` is set to point to the first character which is not converted (unless `endptr` is null).

If no conversion could be performed or if the string is empty, zero is returned, and the initial value of `nptr` is stored in the object pointed to by `endptr` (unless `endptr` is null). If the value is out of range, `+HUGE_VAL` or `-HUGE_VAL`, depending on the sign of the value, is returned. If the value causes underflow, zero is returned. In both these cases, `errno` is set to `ERANGE`.

`strtok` SA T2 break strings into tokens

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

`strtok` breaks the string pointed to by `s1` into tokens, each of which is delimited by a character from the string pointed to by `s2`. The first use of `strtok` must have `s1` pointing at a string. Subsequent use can either have `s1` pointing at a new string or a null pointer as its first argument. If a null pointer is used, the function starts from the position the last call terminated. `s2` can be different for each call. The function returns a pointer to a token or a null pointer if there is no token found.

`strtol` SA convert string to long int

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr,
               int base);
```

This function converts the initial portion of the string pointed to by `nptr` to `long int` representation. First the string is split into three parts: an *initial string* of white-space characters (which may be empty), a *subject string* resembling an integer, to be decoded

using the radix information specified in **base**, and a *final string* which starts at the first character which is not acceptable in the expected format of the subject string, and extends to and includes the terminating NUL character of the input string. Then it attempts to convert the subject string to an integer, and returns the result.

If the value of **base** is in the range 2–36, the expected form of the subject string is a sequence of digits and letters representing an integer with the radix specified in **base**. The letters ‘a’ to ‘z’ (or ‘A’ to ‘Z’) are ascribed the values 10–35. Only those characters which are representations of values less than **base** are allowed. If **base** has the value 16, the characters “0x” (or “0X”) may precede the sequence of letters and digits, but have no effect.

If the value of **base** is 0, the subject string is treated as hexadecimal (if it starts with “0x” or “0X”), octal (if it starts with ‘0’) or decimal (for any other case). All other values of **base** are illegal.

Uppercase letters are everywhere equivalent to lowercase ones, and the subject string may start with a plus or minus sign. However, suffixes (like ‘L’ or ‘U’) are not allowed.

The function attempts to 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 zero 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 equal to **NULL**); in all other cases, including overflows, this object is set to the address of the start of the final string. The subject string will be empty if, for example, the input string is empty or contains only white space. Here are some other input strings whose subject strings are empty:

```
-  
+  
0x  
/
```

```
-00001
0x-5
```

**strtoul** [SA] convert string to unsigned long int

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr,
                        char **endptr, int base)
char *nptr, **endptr;
int base;
```

This function operates in the same way as **strtoul**, except:

- It returns an **unsigned long int**;
- In the event of an overflow being trapped, the value returned is always **ULONG\_MAX**.

**strxfrm** [NEW] [SA] [T2] transform string using locale's collating sequence

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2,
               size_t n);
```

The function transforms the string pointed to by **s2** and places the result in the string pointed to by **s1**. The nature of this transformation is controlled by the **LC\_COLLATE** category of the current locale, and the effect is that two strings which have been transformed in this way can be correctly compared using **strcmp**. A maximum of **n** characters is transformed, including the final null character; in any case, transformation stops after a null character has been converted. **strxfrm** returns the number of characters which have been transformed, excluding the null character. The **s1** argument may be **NULL** if **n** is zero.

Note that, as the current version of Parallel C only supports the "C" and "" locales, this function simply performs a copy operation.

**system** call command interpreter

```
#include <stdlib.h>
int system(const char *string);
```

**string** is passed to the host command-line interpreter and executed as if it had been entered as a command. The string argument to the **system** function should be a valid host command line.

**system** returns 0 if the server accepts the command; otherwise it returns a nonzero value. Any host return code generated by the command is not passed back to the calling program.

**CAREFUL!** Note that it is not normally possible to use a host command which involves the use of the transputer system. Attempting to do this will normally result in the program calling **system** being overwritten by the requested program: when the requested command terminates, the server associated with the original program will not be able to communicate with it and will probably appear to “hang”.

**DOS** Remember that the backslash character ‘\’ used in MS-DOS file names is an escape character in C string literals and must be written as “\\”. For example:

```
system("dir \\mydir\\*.c");
```

**tan** **SA** tangent function

```
#include <math.h>
double tan(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` SA hyperbolic tangent function

```
#include <math.h>
double tanh(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` SA T2 create a simple thread

```
#include <thread.h>
char *thread_create(void (*fn)(), int wssize,
                    int nargs, ...);
```

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 (using `par_malloc`) 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` argument specifies the number of arguments which are to be passed to the new thread's function `fn`. The arguments themselves follow `nargs`. When counting the arguments for `nargs`, each argument of whatever type counts for 1, except for arguments of type `double`, which counts for 2.

This function is a shorthand way of calling the more general thread creation function `thread_start` in the most usual circumstances.

CAREFUL! Because `thread_create` calls `par_malloc` to allocate the workspace from the heap, and because `par_malloc` in turn waits for the `par_sema` semaphore to protect its access to the common data structures in the heap, a thread which calls `thread_create` should *not* have claimed the `par_sema` semaphore. If it has, the call to



`thread_create` will never return because the second wait request for `par_sema` will block unnecessarily.

`thread_deschedule` INLINE SA T2 make current thread momentarily unable to execute

```
#include <thread.h>
void thread_deschedule(void);
```

This function causes a thread to become momentarily unable to execute (usually for one timer tick); this will cause it to be *de-scheduled* 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` INLINE SA T2 return current thread’s priority

```
#include <thread.h>
int thread_priority(void);
```

This function returns the priority of the current thread, which will be either `THREAD_URGENT` or `THREAD_NOTURG`.

`thread_restart` INLINE SA T2 restart a thread given its workspace

```
#include <thread.h>
void thread_restart(char *p);
```

`p` should be a pointer to the workspace of the thread which we wish to restart. Currently, the only value that should be passed to `thread_restart` is one produced by `chan_reset`.

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` SA T2 start a general thread

```
#include <thread.h>
void thread_start(void (*fn)(), char *ws, int wssize,
                  int flags, int nargs, ...);
```

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` argument specifies the number of arguments which are to be passed to the new thread's function `fn`. The arguments them-

selves follow **nargs**. When counting the arguments for **nargs**, each argument of whatever type counts for 1, except for arguments of type **double**, which counts for 2.

The workspace supplied as the **ws** parameter must have been allocated by the caller of **thread\_start**. It may have been allocated in any of a number of ways, including allocation from the heap using **malloc**, or allocation as a simple static or automatic array variable.

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** INLINE SA T2 stop the current thread

```
#include <thread.h>
void thread_stop(void);
```

This function stops the current thread. The current thread is also stopped if its main function returns.

**time** DOS returns the current calendar time

```
#include <time.h>
time_t time(time_t *timer);
```

The **time** function determines the current calendar time. The type (**time\_t**) of the value returned by **time** is **int**. The value returned is the number of seconds that have elapsed since 00:00:00 GMT on 1st January, 1970, according to the host system clock.

If the **timer** argument is not a null pointer, the result of **time** is also assigned to the variable pointed to by **time**. Therefore, the **time** function can be used in either of two ways, as shown in the following

example, where the two statements each assign the current calendar time to the variable `t`:

```
t = time((time_t *)0);
(void)time(&t);
```

**DOS** Although the PC software on which `time` depends attempts to give you the time in GMT, by default it does this on the assumption that you are in the Pacific Standard Time zone. If you are not actually in this time zone, you must make the system aware of the fact. This is done by defining the MS-DOS environmental variable `TZ`. For example, if you live in Great Britain, you could define `TZ` like this:

```
C>set tz=GMT
C>set tz=GMT1BST           (during Summer Time)
```

`timer_after` **INLINE** **SA** **T2** compare two transputer timer values

```
#include <timer.h>
int timer_after(int t1, int t2);
```

This function returns non-zero if timer value `t1` is *after* timer value `t2`, and zero otherwise.

`timer_delay` **INLINE** **SA** **T2** delay for some number of timer ticks

```
#include <timer.h>
void timer_delay(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` INLINE SA T2 return the current timer value

```
#include <timer.h>
int timer_now(void);
```

This function returns the value of the timer associated with the current thread's priority.

For a high priority (“urgent”) thread, the timer has a resolution of  $1\mu\text{s}$ , so that it ticks one million times per second. For a low priority (“not urgent”) thread, the resolution is  $64\mu\text{s}$  and only 15625 ticks occur in one second.

`timer_wait` INLINE SA T2 wait until current timer reaches some value

```
#include <timer.h>
void timer_wait(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`.

`tmpfile` NEW create temporary binary file

```
#include <stdio.h>
FILE *tmpfile(void);
```

This function creates a temporary binary file which will automatically be deleted at the end of the program run. The file is opened for update with `wb+` mode.

**tmpnam** NEW generate unique filename

```
#include <stdio.h>
char *tmpnam(char s);
```

This function generates a unique filename which is not the name of any existing file. Despite the name of the function, a file opened with this name is not automatically deleted at the end of the program run. If the argument **s** is a null pointer, the filename is generated in an internal buffer; otherwise, **s** is assumed to be a pointer to an array of at least **L\_tmpnam** chars, and the filename is written there. The value returned is in both cases a pointer to the place where the filename has been written.

You may call **tmpnam** a maximum of **TMP\_MAX** times, and each time it will generate a different filename. The internal buffer is only guaranteed to remain unchanged until the next call to **tmpnam**.

In the current implementation, **TMP\_MAX** is  $10000_{16}$ , and **L\_tmpnam** is 9. The form of the generated filenames is **tmp\$nnnn**, where *nnnn* is a hexadecimal number.

**tolower** SA T2 convert char to lower case

```
#include <ctype.h>
int tolower(int cval);
```

If **cval** is the ASCII code for an upper case letter, **tolower** returns the code for the corresponding lower case letter. Otherwise, the value of **cval** is returned unchanged.

**toupper** SA T2 convert char to upper case

```
#include <ctype.h>
int toupper(int cval);
```

If **cval** is the ASCII code for a lower case letter, **toupper** returns the code for the corresponding upper case letter. Otherwise, the value of **cval** is returned unchanged.

**to86** DOS transfer memory block to host

```
#include <dos.h>
int to86(int len, char *here, pcpointer there);
```

This function transfers **len** bytes of transputer memory starting at **here** to a corresponding block starting at **there** in host memory. The function returns the number of bytes actually transferred. The host memory block will normally have been allocated by a previous call to **alloc86**.

**ungetc** push character back into input stream

```
#include <stdio.h>
int ungetc(int cval, FILE *stream);
```

**ungetc** pushes the character **cval** back on an input **stream**. That character will be returned by the next **getc** call on that **stream**. **ungetc** returns **cval**.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

**fseek** (q.v.) erases all memory of pushed back characters.

**ungetc** returns **EOF** if it can't push a character back.

**va\_arg** MACRO NEW SA T2 access next argument in  
variable-length list

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

The **va\_arg** macro is used to access the next argument in a variable-length argument list. The parameter **ap** should be a variable of the type **va-list**, which is defined in the **<stdarg.h>** header; it must have been initialised by **va\_start**. The macro expands into an expression which has the value of the next argument and the specified type; if this is not in fact the type of the argument, or if there are no more arguments, the behaviour is undefined. For example:

```
#include <stdarg.h>
void ourfunc (char *message, ...);
{
    va_list ap;
    int ival;
    va_start (ap, message);
    ival = va_arg (ap, int);
}
```

On each call of **va\_arg**, the parameter **ap** is modified to point to the next argument in the list.

**va\_end** MACRO NEW SA T2 finish with variable-length argument  
list

```
#include <stdarg.h>
void va_end(va_list ap);
```

When accessing a variable length argument list, a function should call this macro once all the arguments have been processed. This ensures a correct return to the calling function. The parameter **ap** should be a variable of the type **va-list**, which is defined in the **<stdarg.h>** header; it must have been initialised by **va\_start**.



**va\_start** MACRO NEW SA T2 initialise argument pointer

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

**va\_start** is called before accessing a variable-length argument list. The parameter **ap** should be a variable of the type **va\_list**, which is defined in the **<stdarg.h>** header. The parameter *parmN* should be the parameter in the variable-argument list immediately before the “, ...”.

**vfprintf** NEW formatted output using argument pointer

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, char *format,
             va_list ap);
```

This function corresponds to **fprintf**, and performs formatted output to the specified **stream**. As with **fprintf**, the **format** argument controls the conversions to be performed. However, the variable argument list has been replaced by the single argument **ap**, which should be an argument pointer initialised by **va\_start**. For example:

```
#include <stdarg.h>
#include <stdio.h>
void error (char *func_name, char *format, ...)
{
    va_list ap;
    va_start (ap, format);
    fprintf (stderr, "Error in %s: ", func_name);
    vfprintf (stderr, format, ap);
    va_end(ap);
}
```

The function returns the number of characters output, or a negative value if an output error occurred.

`vprintf` NEW formatted output to `stdout` using argument pointer

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(char *format, va_list ap);
```

This function corresponds to `printf`, and performs formatted output to the standard output stream, `stdout`. As with `printf`, the `format` argument controls the conversions to be performed. However, as with `vfprintf`, the variable argument list has been replaced by the single argument `ap`, which should be an argument pointer initialised by `va_start`.

The function returns the number of characters output, or a negative value if an output error occurred.

`vsprintf` NEW SA formatted output to a string using argument pointer

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *pstr, char *format,
             va_list ap);
```

This function corresponds to `sprintf`, and writes formatted output into a character array via a pointer `pstr` supplied by the user. As with `sprintf`, the `format` argument controls the conversions to be performed. However, as with `vfprintf`, the variable argument list has been replaced by the single argument `ap`, which should be an argument pointer initialised by `va_start`.

The function returns the number of characters output, or a negative value if an output error occurred.

**wcstombs** NEW SA T2 convert wide character string to multibyte string

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs,
                 size_t n);
```

The sequence of wide characters pointed to by **pwcs** is converted to a multibyte string and stored in the array pointed to by **s**. Conversion stops when a null character has been converted, or when the next character stored would exceed the limit of **n** bytes. If the two string overlap, the effect is undefined.

**wcstombs** returns the number of bytes stored, excluding the null character, if any.

Note that, in the present version of Parallel C, multibyte characters and wide characters are both one byte in length and there is no state-dependent encoding, so this function is equivalent to a string copy. All possible element values are valid, so no error return can happen.

**wctomb** NEW SA T2 convert multibyte character to wide character

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

If **s** is a null pointer, **mbtowc** returns 0, indicating that, for the current version of Parallel C, multibyte character codings are never state dependent. Otherwise, it returns the width in bytes of the multibyte character corresponding to value of **wchar**. In the current version, this will always be 1.

In addition, the multibyte character corresponding to the value of **wchar** will be stored at the location pointed to by **s**. In the current version, as both wide and multibyte characters are always 1 byte in length, this is equivalent to storing **wchar** at **\*s**.



# Chapter 12

## The Linker

The linker utility, `linkt`, is compatible with all versions of the 3L compilers for C, Fortran and Pascal. It can be used in place of the linker distributed with earlier versions of these compilers. The linker is also compatible with Tbug, 3L's interactive source-level debugger.

The linker's function is to create an executable file from a number of object files. It can also be used to create libraries of object modules, which may themselves be searched by the linker when it is creating executable files.

### 12.1 Command Line

The linker is invoked by the command `linkt`. This command is followed by an ordered list of items giving the names of the object files and libraries to be linked, the name to be used for the executable file, and switches to control the linking operation.

The name of the executable file must be separated from the object file names by a comma `,`; each object file may be separated from the next by either a space or a plus sign, `+`. Switches all start with

a slash, '/', and so do not need to be separated one from another, but spaces may be inserted between them for clarity.

The following are all valid examples of link commands.

```
C>linkt prog.bin library.bin,prog.b4
C>linkt prog.bin+library.bin, prog.b4
C>linkt prog1.bin prog2.bin lib.bin, myprog.b4
C>linkt prog1.bin+prog2.bin+lib.bin, myprog.b4
C>linkt prog.bin lib.bin, myprog.b4 /Q/Smyprog.map/0kernel
C>linkt prog.bin lib.bin, myprog.b4 /Q /Smyprog.map /0kernel
```

The order of the object file names in the link command is used to order the placement of the information they contain in the resulting executable file. Often this ordering is of no interest but it can be used to improve the performance of programs. This subject is discussed further in section 12.6.

## 12.2 File Name Conventions

In order to simplify commands, the linker will insert file name extensions where none has been given. If an explicit extension has been given it will be used without change.

The actual extension that will be appended to a file name depends on the sort of file being identified. The following table gives each sort of file known to the linker along with the appropriate extension.

executable file	.b4	object file	.bin
indirect file	.dat	optimization file	.opt
input library file	.bin	output library file	.lib
map file	.map		

As a result the examples given previously would have the identical effect if written in the following ways:

```
C>linkt prog library,prog
C>linkt prog+library, prog
C>linkt prog1 prog2 lib, myprog
```

```
C>linkt prog1+prog2+lib, myprog
C>linkt prog lib, myprog /Q/Smyprog/0kernel
C>linkt prog lib, myprog /Q /Smyprog /0kernel
```

## 12.3 The Output File

The output from a linking operation is usually a file containing a complete program in a form ready for execution. This file is called an *executable file*. The output may also be a *library* suitable for input to a subsequent link operation. Section 12.5 describes libraries.

The name for the output file is either specified explicitly on the command line (as in all the examples so far) or is inferred by the linker from the name of the *first* object file (or library file) seen, by removing any extension and then appending the extension `.b4`.

For example, each of the following commands generates an executable file named `test.b4`:

```
C>linkt test.bin fns.bin lib.bin, test.b4
C>linkt test fns lib, test
C>linkt test fns lib
C>linkt fns test lib, test
```

## 12.4 Indirect Files

It is quite common for programs to be built from a large number of object files, perhaps more than can comfortably be fitted into a single `linkt` command line.

The linker addresses this problem with *indirect files*, each of which contains one or more file names on separate lines. Indirect files may be given wherever object files are expected and the file names they contain are interpreted as the names of object files to be included in the linking operation.

In linker command lines, indirect files are always marked with the symbol ‘@’ to distinguish them from other sorts of file. It is also possible to mark names within indirect files in this way. Such names are then taken to be the file names of nested indirect files. Indirect files may only be nested to a depth of 5.

For example, assume the file `list.dat` contains the following:

```
file1.bin
file2
file3.xxx
```

In the following example, the first four commands will all have the same effect, while the fifth command will generate an identical executable file but will write it to a file named `prog.b4`:

```
C>linkt @list
C>linkt @list, file1.b4
C>linkt file1 file2 file3.xxx
C>linkt file1 file2 file3.xxx, file1.b4
C>linkt @list, prog
```

Note that in the examples above, the first object file name in the indirect file will be the first object file seen by the linker and so it will be that file name which will be used, if necessary, to deduce the name for the output file.

Indirect files are also used to supply a list of optimization symbols to the linker. This is described in section 12.6.

## 12.5 Libraries

It is often convenient to be able to treat a group of object files as a single unit known as a *library file*. Accordingly, the linker provides the option of combining object files (and library files) into a new library file rather than the more usual executable file.



Once a library file has been generated it may be used wherever an object file is expected; unlike indirect files there is no need to mark the library file name in any way.

Library files have two advantages over indirect files. Firstly, moving a single library file to another place is simpler than moving many component object files and making sure that the corresponding indirect file is kept up to date. Secondly, accessing a single library file is faster than accessing an indirect file and several object files.

During the development of components which will eventually make up a library, indirect files may be more convenient as there will be no need to re-link the library whenever a component object file is changed.

The linker command to create a library is similar to that used to create an executable file, but includes the switch `/L`. When this switch is used the output file will be a library file and not an executable file. The name of the library file will be deduced, if necessary, in the same way as for executable files; that is, from the name of the first object file or library file found. The default extension `.lib` will be added if no extension is given.

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 linker commands.

```
C>type graflib.dat
core.bin
tek.bin
hp.bin

C>linkt @graflib,graflib/L

C>type myprog.dat
myprog.bin
graflib.lib
library.bin
harn.bin
```

```
C>linkt @myprog,myprog.b4
```

The switch `/P` can be used in place of `/L` and has exactly the same meaning.

The following switches are ignored when the `/L` or `/P` switches are used: `/B`, `/C`, `/O`, `/S` and `/X`. Section 12.10 contains a full description of the switches.

If the `/G` option is used when creating a library, any debug information present in the object files is passed through into the library. Otherwise this information is left out of the library.

## 12.6 The Executable Image

Unless otherwise instructed, the linker will place object files it has selected into the executable file in the order in which those object files were specified on the command line. This order is important if a program wishes to make use of the on-chip RAM.

When the on-chip RAM is used to hold programs, the code which has been placed at the beginning of an executable image is more likely to reside in RAM than code towards the end. Hence, in order to improve the performance of a program, the object file containing the code which is executed most frequently should be specified as the first object file in the link command.

In many cases, it may not be easy or possible to know which order to place the object files in. For example, the user may know which functions are executed most frequently, but not know which object files contain them, because they are part of a library. In this case, the user can specify a symbol to search for, and the linker will look for an object file which contains a definition of that symbol. Symbols used like this are known as *optimization symbols*, and are specified by using the `/O` command-line switch. Note that the switch uses the *letter* ‘O’ and not the *digit* ‘0’.

As an example, the following will place the object files which contain definitions of `fread` and `malloc` at the beginning of the resulting executable file `t.b4`:

```
C>linkt t library harn/Ofread/0malloc
```

In this case, the object file containing the external symbol `fread` is placed at the start of the executable image. The object file containing the external symbol `malloc` is placed second in the executable image.

If an optimization symbol does not exist then the linker issues a warning. Sometimes the object file containing the symbol is not needed in the executable image; in other words, there are no references to it. In this case, if the object file is part of a library, the module is excluded from the executable image, and no warning is issued. If, on the other hand, the symbol is found in an object file named in the command line or in an indirect file, the object file is included in the executable image regardless.

Two or more optimization symbols may refer to the same object file, in which case the position of the object file will be determined by the position of the first symbol to refer to it.

After all optimization symbols have been processed and the object files which define optimized symbols have been placed at the start of the executable image, the linker will add the remaining object files to the executable image in the order they were found on the command line. In the previous example this would mean that object file `t` would be the third object file in the executable image and the object file `harn` would be the last.

It is often easier to place the list of optimization symbols in a file rather than keeping them on the command line. This may be done using indirect files in the same way as for object files except that the default extension is now `.opt`.

An example optimization file `optsyms.opt` might contain the following text:

```
fread
```

```
malloc
```

This file could then be used to optimize the position of the object files defining `fread` and `malloc` as in the following command:

```
C>linkt t library harn/0@optsyms
```

A warning is issued if the symbol is not defined in any of the object files.

## 12.7 Map Files

The linker can be requested to produce a *map file* which will contain a list of all the symbols (both code and data) that have been defined in the executable image. The map file will also contain information about the sizes of the code and static areas for each object file.

Map files are requested with the `/S` switch. By default, the name of the map file is derived automatically from the first object file name. In the following example a map file called `test.map` would be generated.

```
C>linkt test library harn/S
```

Alternatively, the map file name can be specified explicitly on the command line by placing a file name immediately after the `/S` as in the following example:

```
C>linkt test library harn/Smfile
```

The default extension `.map` will be added if no extension is given. The above example would create a map file called `mfile.map`.

## 12.8 T2 Support

A special set of linker command-line switches are used to support the use of 16-bit transputers. These switches should *only* be used

when linking code for T2 transputers; they should not be used when linking code for T4 or T8 transputers.

The command-line switches for T2 support are described below.

### 12.8.1 Switch */Msize*

This switch gives the total number of bytes of read-write memory available to the program. The memory will be used to hold the static data, heap and stack for the running program. In addition, it will hold the executable code of the program unless the code is to be held in read-only memory.

You must give a */M* switch when linking for T2 systems unless you intend to control the linker's memory allocation by means of modified */F* switches.

The batch file `t2clink` provides a default value of 64K for this switch (*/M64K*) but you may override this default with another */M* switch of your own, e.g.,

```
C>t2clink main bits pieces/M24K
```

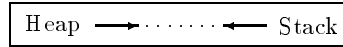
If more than one */M* switch appears on a command line, only the last will have any effect.

The linker will give a warning if you specify less than 2048 bytes or more than 65536 bytes of read-write memory.

### 12.8.2 Switch */Asize*

This switch controls the number of bytes of read-write memory to be used for the stack ("automatic" storage in C terminology). The linker will give a warning if you specify less than 128 bytes of stack. Memory for the stack is taken from the read-write memory remaining after the code and static data areas have been allocated.

If you do not specify this switch then the whole of the remaining memory will be used for a combined heap and stack area. The stack will grow towards the heap from the more positive end of the area while the heap will grow towards the stack from the more negative end of the area.



If you do give a `/A` switch, the given amount of memory will be allocated to the stack and the whole of the remaining memory will be used for the heap. In this case the stack and heap areas will be considered distinct and will not interact.

### 12.8.3 Switches `/FC`, `/FA`, `/FS`, and `/FH`

These switches are used to control the order in which the various areas of the program are loaded into the available memory: `/FC` for code, `/FA` for the stack (automatic) area, `/FS` for static data, and `/FH` for the heap.

The linker will usually construct an executable image by laying out the various areas (code, static data, heap, and stack) in memory, starting at the most negative address usable—in the fast, on-chip memory. Consequently the parts of the image which are placed first will benefit from the speed of this memory.

The `/F` switches give you control over the order in which the areas will be laid out. Any area mentioned in a `/F` switch will be considered a candidate for “optimisation”—you can think of the ‘F’ as standing for “fast”. For example, the switches `/FC/FS` indicate that the code and static data areas are to be optimised. The order in which you give the `/F` switches is of no significance.

The linker will lay out all of the optimised areas before it lays out any non-optimised areas. The order in which areas (optimised or not) are laid out depends on the presence or absence of the `/A` switch.

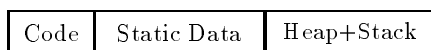
If you do not specify the `/A` switch, then the stack and heap areas will be combined, as described above. In this case the linker will lay out the areas in the order: code, static data, and then the combined stack and heap.

If you do specify the `/A` switch, then the stack and heap areas will remain distinct, and the linker will lay out the areas in the order: stack, code, static data, and then heap.

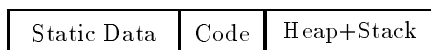
The following pictures should clarify this procedure. Note that in these pictures addresses grow more positive towards the right hand side.



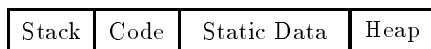
`C>linkt prog`



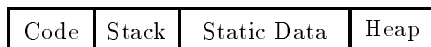
`C>linkt prog/FS`



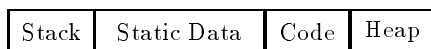
`C>linkt prog/A8K`



`C>linkt prog/A8K/FC`



`C>linkt prog/A8K/FS/FA`



The system described is designed to allow the most common requirements to be specified simply.

### 12.8.4 Modified /F Switches

The /F switches may be modified so that instead of simply marking areas for optimisation they explicitly specify the memory locations to be used.

To modify the switches you append an address and size specification of the form *start:size*, where *start* is the address for the start (smallest address) of the area and *size* is the size of the area in bytes. If *start* or *size* begin with a '#' character they will be interpreted as hexadecimal, otherwise they will be interpreted as decimal. All values of *start* and *size* must be even. Note that the start address of the stack area is *not* the initial value for Wptr; that value will be *start + size*. For example:

```
C>linkt x y z /FC#1000:80 /FH#2000:#2000 /FA#8000:4096 /FS0:8
```

The linker will check that these areas do not overlap and issue an error message if they do. Similarly, the linker will issue an error message if either the code area or the static data area is too small for the linked image. The total size of the static data area for a task will be:

$$2 \times \text{modules} + \sum_{i=1}^{\text{modules}} \text{static}_i$$

There are several implications of modifying /F switches in this way:

1. If you specify one modified /F switch then you must specify and modify all four. The only exception to this rule is when you are linking for ROM (described later);
2. There will be no automatic optimisation or memory allocation. Memory allocation is fully under your control;
3. The stack and heap areas will be considered separate, even though they may be adjacent. This means that while the program is running the heap will never extend into the stack area.



### 12.8.5 Switch */Rsize*

The */R* switch instructs the linker to generate an image suitable for burning into ROM. The image size will be exactly the number of bytes specified in the */R* switch.

When a ROM program starts execution, it copies its static data from the ROM into read-write memory.

The code may either be left in the ROM or copied into the read-write memory. This is controlled by the */FC* switch. If no */FC* switch is specified then the code will be executed from the ROM. If a */FC* is specified (modified or not) then the code will be copied into the read-write memory before being executed.

Note that when linking for ROM with modified */F* switches you may omit the modified */FC* switch if you wish the code to be executed from ROM. Of course, you should make sure that none of the areas overlaps any ROM addresses.

## 12.9 Debug Tables

Object files created using the 3L compilers may contain information intended for use by Tbug, the 3L debugger. By default, the linker will discard this information in order to produce small executable files.

The switch */G* will make the linker incorporate any debugging information present in the object files into the output file, which may be either an executable file or a library file.

## 12.10 Summary of Switches

The operation of the linker can be controlled by means of *switches*. Each switch starts with a slash character *'/'* and an identifying letter;

it does not matter if this letter is given in upper case or lower case. The switches can be placed anywhere in the command line but they may not occur in indirect files. No spaces are allowed between a switch's identifying letter and the rest of the switch.

<i>/Asize</i>	This switch defines the size of the stack area. This should only be used when linking code for T2 transputers.
<i>/Bfile-name</i>	This switch specifies that the file <i>file-name</i> is to be used in preference to the default bootstrap file. There is no default extension for <i>file-name</i> .
<i>/C</i>	This switch stops the linker adding the bootstrap file to the executable file.
<i>/FA</i>	This switch instructs the linker to optimise the stack (automatic) area. This should only be used when linking code for T2 transputers.
<i>/FAstart:size</i>	This switch defines the <i>start</i> address and <i>size</i> of the stack area. This switch should be used along with the other modified <i>/F</i> switches and should only be used when linking code for T2 transputers.
<i>/FC</i>	This switch instructs the linker to optimise the code area. This should only be used when linking code for T2 transputers.
<i>/FCstart:size</i>	This switch defines the <i>start</i> address and <i>size</i> of the code area. This switch should be used along with the other modified <i>/F</i> switches and should only be used when linking code for T2 transputers.
<i>/FH</i>	This switch instructs the linker to optimise the heap area. This should only be used when linking code for T2 transputers.

*/FHstart:size*

This switch defines the *start* address and *size* of the heap area. This switch should be used along with the other modified */F* switches and should only be used when linking code for T2 transputers.

*/FS*

This switch instructs the linker to optimise the static data area. This should only be used when linking code for T2 transputers.

*/FSstart:size*

This switch defines the *start* address and *size* of the static data area. This switch should be used along with the other modified */F* switches and should only be used when linking code for T2 transputers.

*/G*

This switch results in the linker creating a debugger information area in the executable or library file. This switch should not be used when linking code for T2 transputers.

*/I*

This switch causes the linker to display its identity and along with various statistics about the executable file such as the code and static sizes and the maximum patch size used.

*/L*

This switch makes the linker generate a library file rather than an executable file.

*/Msize*

This switch specifies the size of read-write memory area (including on-chip memory.) This should only be used when linking code for T2 transputers.

*/Ooptimization-symbol*

This switch gives priority to the position in the executable image of the object file which defines *optimization-symbol*.

***/O@optimization-file***

This switch gives priority to the position in the executable image of the object files which define the symbols whose names are contained in the file *optimization-file*. The default extension for *optimization-file* is *.opt*.

***/P*** This switch has the same effect as the */L* switch.

***/Q*** This switch suppresses all warning messages (see section 12.13).

***/Qn*** This switch suppresses every occurrence of warning message number *n* (see section 12.13).

***/Rsize*** This switch gives the size of the read-only memory area. This should only be used when linking code for T2 transputers.

***/S*** This switch generates a map file taking its name from the first name in the list of object files.

***/Smap-file*** This switch generates a map file called *map-file*. The default extension for *map-file* is *.map*.

***/Xentry-point***

This switch causes the linker to use the symbol *entry-point* in preference to `INMOS.ENTRY.POINT`, which is the default.

## 12.11 Using Batch Files

A batch file is a convenient way of calling the linker with the appropriate run-time library and harness. The linker accepts spaces between object file names, so the batch file can pass more than one parameter to the linker; unused parameters will be ignored. Switches can appear in any position on the command line, so they can be

passed as parameters to the batch file. For example, the batch file `mlink.bat` might look like this:

```
C>linkt %1 %2 %3 %4 %5 %6 %7 %8 %9 library.bin harn.bin
```

The following example shows how the batch file could then be used to link two files `file1.bin` and `file2.bin` into a library `file1.lib`:

```
C>mlink file1 file2/L
```

The batch file will then invoke the linker with the following command:

```
linkt file1 file2/L library harn
```

It is not possible to include a comma in a batch file parameter. For this reason, you cannot explicitly pass an output file name to a batch file in its command line.

## 12.12 Duplicate Definitions

A duplicate definition occurs if two or more object files define the same symbol. The linker will issue a warning message about each occurrence of a duplicate definition and will use the first definition encountered. Object files are processed in the order in which they appear on the command line.

This facility can be useful when it is necessary to rewrite or alter an object file contained in a library. It can also be used to substitute one object file for another when creating a new library.

Occasionally, for example when several libraries are being used, it may be desirable to suppress a very large number of duplicate definition warning messages. This can be done by using the switch `/Q1`. This facility may be useful for OEM users of the linker.

## 12.13 Messages

The linker may issue one or more messages during a linking operation. These messages are used to draw the user's attention to unusual or incorrect situations.

There are two types of message: *warnings*, which indicate acceptable but possibly erroneous conditions, and *fatal errors* which result from conditions which are serious enough to terminate the linking operation. The `/Q` switch may be used to suppress all warning messages; the form `/Qn` can be used to suppress all occurrences of message number *n*.

In order to give as much useful information as possible, the linker will often expand messages by including such things as symbol names and numerical values. In the description of the messages, terms in *italics* will be replaced appropriately according to the following scheme:

<i>filename</i>	The name of a file, such as an object file name
<i>module</i>	The name of an object file or a module within a library
<i>number</i>	An integer value
<i>switch</i>	A letter used to identify a command-line switch
<i>symbol</i>	A symbol defined or referenced by an object file or module
<i>text</i>	Various pieces of descriptive or illustrative text
<i>type</i>	A code for a specific type of transputer, such as T414

WARNING (0): data symbol *symbol* referenced as a code symbol in *module*

Description The given *symbol* has been defined as a *data symbol* but the given module references it as a *code symbol*.

User Action Check that the code symbol references are specifying the correct symbol and that the symbol has been defined appropriately.

WARNING (1): using definition of *symbol* in *module1*,  
ignoring duplicate in *module2*

Description The given *symbol* has already been defined by *module1*. Another definition has subsequently been found in *module2*. This latter definition will be ignored.

User Action Check that the correct definition is being used. If the second definition was the one that was really wanted, change the order of the object files in the link command so that the file (or library) containing the wanted definition comes before the unwanted definition.

FATAL ERROR (3): multiple INIT tags

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (4): multiple MAININIT tags

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (5): object file *filename* is corrupt; illegal patch/*number*

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (6): object file *filename* is corrupt; unknown tag/*number*

Description This error indicates that an object file is internally inconsistent.

User Action Check that the files being linked together are proper object files or libraries.

FATAL ERROR (7): incompatible processor types; *type1* in *module1* and *type2* in *module2*

Description Code compiled for one type of transputer may not be able to execute correctly on a different type of transputer. This error indicates that object files compiled for a processor of *type1* are being linked with object files compiled for a processor of *type2*.

User Action Decide the type of the target processor and recompile those object files which had been compiled for a different processor type. Also check that the correct run-time library has been specified.



FATAL ERROR (8): reserved symbol *symbol* defined in *module*

Description The linker reserves certain symbols for its own use. This error indicates that the given module has attempted to define such a symbol. The reserved symbols all start with two consecutive underline characters. Users should avoid using any symbols which start with these characters.

User Action Avoid using the reserved symbol.

FATAL ERROR (9): internal error/*number*

Description This message is issued when the linker discovers that its internal tables are in an inconsistent state.

User Action Submit a fault report to your distributor including the exact text of the error message.

WARNING (10): module *module* refers to undefined symbol *symbol*

Description The given module contains a reference to the given symbol. By the time all of the object files and libraries given in the command line have been examined, no module has been found which contains a definition of the symbol. Although this is really a fatal error, it is treated as a warning so that further undefined symbols may be discovered and reported. A fatal error (47 or 48) will be issued on the completion of the link operation.

User Action Check that the module which defines the symbol has been included in the link command and that the name

of the symbol has been given correctly in both the module which defines it and the module which references it.

**FATAL ERROR (11): multiple main static initialization modules**

**Description** This error indicates that an object file is internally inconsistent.

**User Action** Check that the files being linked together are proper object files or libraries.

**FATAL ERROR (12): entry point symbol *symbol* has not been defined**

**Description** The given symbol has been specified as the entry-point for the program being linked, either by default (in which case the symbol will be `INMOS.ENTRY.POINT`) or explicitly using the `/X` linker switch. This error indicates that no module has defined that symbol.

**User Action** Check that the entry-point symbol has been specified correctly in the `/X` switch or that the list of object files to link includes one which defines the main entry point (a C `main` function, a Fortran `PROGRAM`, or a Pascal `PROGRAM`).

**WARNING (13): no definition found for optimization symbol *symbol***

**Description** The given symbol has been nominated for optimization by means of the `/O` linker switch. The warning is issued if no definition for that symbol has been found by the end of the linking operation.

User Action Check that the symbol has the correct spelling and that the module containing its definition has been included in the list of files to be linked.

**WARNING (14): cannot optimize position of debug area**

Description It is not possible for the linker to optimize the position of the debug area. This area is identified by means of a reserved symbol, `__debug_area`, which is defined by the linker itself. The warning is issued if the name of the debug area is nominated in a `/O` linker switch.

User Action Remove the debug area symbol from the list of optimization symbols.

**FATAL ERROR (15): no MAININIT found: language run-time library missing?**

Description This error indicates that the linker has been unable to find the definition of an initialization module which is assumed by the 3L compilers. The problem is usually caused by omitting the run-time library from the list of files to be linked.

User Action Include the appropriate run-time library in the list of files to be linked.

**FATAL ERROR (18): code position exceeds declared size in module**

Description This error is usually caused by an error during the compilation of the given module.

User Action Recompile the given module and attempt the link operation again. If the fault persists, please contact your distributor.

**FATAL ERROR (21): not enough memory**

**Description** This error indicates that the linker has run out of available memory.

**User Action** If the memory available on the transputer board is more than 2MB use the `mempatch` program on the linker to set the correct memory size for the linker. The `mempatch` program is described in the 3L Parallel language manuals.

**FATAL ERROR (22): patch over valid code in module *module***

**Description** When compiling the instructions used to access external objects, the compiler leaves a certain amount of room for the linker to *patch* in the actual address of the object. This error indicates that the size of the image file is such that the space left in the given module is not big enough to hold the actual address.

**User Action** Recompile the offending module and increase the amount of space left by the compiler for patches using the appropriate compile-time switch. Refer to the compiler's documentation for details.

**FATAL ERROR (23): internal limitation -- too many references (*number*)**

**Description** The linker cannot complete the linking operation because the files being linked have more references to external symbols than can be held in the linker's internal tables. The linker's tables have space for approximately 150,000 external references.

**User Action** The only appropriate action is to reduce the number of external references by concatenating some of the original source files into a single file.

**FATAL ERROR (24): internal limitation -- too many common blocks (*number*)**

**Description** The linker's internal tables used for describing definitions of and references to Fortran common blocks have been filled and so the linker cannot complete the linking operation. The linker allows a total of approximately 64,000 such references and definitions.

**User Action** The only appropriate action is to attempt to concatenate some of the source programs that reference the same common blocks. For example, if five source programs refer to common block **X** then concatenating those files into a single source file and recompiling will reduce the number of references to common blocks from five (one per original source file) to one (in the resulting combined file).

**FATAL ERROR (25): internal limitation -- unexpected end of file/*number***

**Description** This error is the result of attempting to link a corrupt object file or a file which is not an object file.

**User Action** Check that all of the files being linked are object files.

**FATAL ERROR (26): internal limitation -- vector size (*number*) exceeds limit (*number*)**

**Description** This error indicates that a record in the object file exceeds the maximum size allowed.

**User Action** Check that all files being linked together are proper object files or libraries.

FATAL ERROR (27): internal limitation -- too many optimization symbols (*number*)

Description The linker can only process a limited number of optimization symbols. The linker allows for approximately 1024 symbols. This error indicates that too many optimization symbols have been specified.

User Action Remove some optimization symbols.

FATAL ERROR (30): all object files are libraries; nothing to link

Description This error indicates that all of the files given on the command line are library files.

User Action Add an object file to the list of library files, or, if you mean to generate a new library file, use the */L* switch.

FATAL ERROR (31): unknown processor type *type* in *module*

Description The given module has indicated that the code it contains is for a transputer of the given type. This type does not correspond to a transputer known to the linker.

User Action Recompile the offending module, specifying a known transputer type.

FATAL ERROR (32): unable to write to file *filename*

Description This error indicates that the named file cannot be created successfully.

User Action Check that the file name has been specified correctly and that the device on which the file is to reside has enough free space.

FATAL ERROR (33): internal limitation -- cannot process more than *number* object files

Description The linker can only process the limited number of object files and libraries. The linker allows for approximately 16,000 files. This error indicates that too many object files and libraries have been specified.

User Action Combine some of the object files into a single library file and use that library instead of the individual files.

FATAL ERROR (34): unable to open *filename*

Description This error is issued when the linker is unable to access a file.

User Action Check that the given file exists and that its name has been specified correctly.

FATAL ERROR (35): internal limitation -- too many modules (*number*)

Description The linker can only process a limited number of modules. The linker allows for approximately 16,000 modules. The error indicates that too many modules have been specified.

User Action Combine individual modules together at the source code level.

FATAL ERROR (36): internal limitation -- too many symbols (*number*)

Description The linker can only process a limited number of symbols. The linker allows for approximately 128,000 symbols. The error indicates that too many symbols have been specified.

User Action Remove any unnecessary external symbols from the source programs.

FATAL ERROR (37): command line: *text* expected

Description This error indicates that the command line has been incorrectly formed.

User Action Correct the command line.

FATAL ERROR (38): cannot specify output file twice

Description This error indicates that two or more commas have been found on the linker command line. A comma is used to separate the name of the linker's output file from the files to be linked, and there may only be one such output file.

User Action Check the format of the linker command line. In particular, make sure that the list of object files does not include commas.

FATAL ERROR (39): option */switch* not recognised

Description The given switch is not a linker option.

User Action Correct the specification of the option.

FATAL ERROR (40): internal limitation -- too many nested data files

Description The linker imposes a limit on the depth to which data files (indirect files) can be nested. Currently, this limit is 5.



User Action Replace the most deeply nested references to indirect files with their contents.

FATAL ERROR (47): 1 symbol undefined

Description This error is issued at the end of a linking operation in which a single “undefined symbol” warning was produced

User Action Refer to WARNING (10)

FATAL ERROR (48): *number* symbols undefined

Description This error is issued at the end of a linking operation in which several “undefined symbol” warnings were produced.

User Action Refer to WARNING (10)

WARNING (51): object file *filename* is corrupt; missing T2 items

Description This error should not occur.

User Action Submit a fault report to your distributor including the exact text of the error message

WARNING (52): command line must specify memory size when linking T2 objects

Description This error is issued when no /M option is given when linking code for T2 transputers.

User Action Specify the memory size using the /M option on the linker command line. This size should be between 2KB and 64KB.

**FATAL ERROR (53): memory size specified (*number*) is less than 2048 bytes**

**Description** This error indicates that, when linking for T2 transputers, the /M option was used to specify a memory size smaller than the minimum of 2048 bytes.

**User Action** Use a larger value with the /M option switch on the linker command line.

**FATAL ERROR (54): memory size specified (*number*) is greater than 64K bytes**

**Description** This error indicates that the memory size given, using the /M option when linking for T2 transputers, is greater than the address space of a 16-bit transputer.

**User Action** Use the /M option to specify a memory size between 2KB and 64KB.

**FATAL ERROR (55): stack size specified (*number*) is less than 128 bytes**

**Description** This message indicates that, when linking for T2 transputers, the size specified for the stack area, using either the /A or the /FA option switch, was less than the minimum or 128 bytes.

**User Action** Change the option parameter to allow a larger stack.

**FATAL ERROR (56): debug table generation disabled when linking T2 objects**

**Description** This message is issued when /G option was used when linking code for T2 transputers. A fatal error (9) will be issued on the completion of the link operation

User Action Remove the `/G` option from the linker command line.

**WARNING (59): image is larger than memory size specified  
(*number*)**

Description The size of the image exceeds the memory size specified with the `/M` option when linking code for T2 transputers.

User Action Specify a larger memory size—this should of course correspond to the memory size available on the target transputer.

**FATAL ERROR (60): area positions specified on command  
line overlap**

Description This indicates that when linking code for T2 transputers the modified `/F` switches have been used to define memory areas which overlap each other.

User Action Check the start address and size of each memory area to determine which are overlapping. Change the start address or size of one or more of the memory areas so that there are no overlaps.

**FATAL ERROR (61): `/a` cannot be used if area positions  
are specified**

Description The modified `/F` switches have been used with the `/A` switch on the command line when linking code for T2 transputers.

User Action When the modified `/F` switches are used the `/A` switch can be omitted. The `/FA` switch can be used to specify the start address and size of the stack area.

FATAL ERROR (62): /m cannot be used if area positions are specified

Description The modified /F switches have been used with the /M switch when linking code for T2 transputers.

User Action When the modified /F switches are used to specify each of the memory areas the /M switch should be omitted.

FATAL ERROR (63): all area positions must be specified

Description One or more of the modified /F switches has been omitted from the command line.

User Action Supply all four of the switches, for a further description of these see section 12.8.4.

FATAL ERROR (64): all area positions and sizes must be word aligned

Description The modified /F switches used define a memory area which does not begin and end on word boundaries. This message only occurs when linking code for T2 transputers.

User Action Check that the start addresses and sizes of all the memory areas are even.

WARNING (65): code size specified is smaller than actual code size of *number* (hex) bytes

Description This message indicates that the space required for the program code exceeds the size of the memory set aside for code using the /FC switch.

User Action Increase the size of the code area specified with the /FC switch.

WARNING (66): data size specified is smaller than actual data size of *number* (hex) bytes

Description This message indicates that the static data requirement of the program being linked exceeds the memory area set aside for static data using the /FS switch.

User Action Increase the size of the static data area specified with the /FS switch.

FATAL ERROR (67): *memory* area overlaps ROM area

Description The memory region specified by one of the modified /F switches overlaps the ROM area, defined with the /R switch. *memory* will be one of **static data**, **stack**, **code**, or **heap**.

User Action Change the size or start address of the memory area that overlaps the ROM. The size of the ROM determines its start address since its last address will be FFFF.

FATAL ERROR (68): no space in heap or stack area for bootstrap of length *number* bytes

Description The memory set aside for the stack is too small to hold the bootstrap code.

User Action The bootstrap used for T2 transputers is usually placed in the memory that will be used for stack and heap when the program has been booted. In this case the space is too small to hold the bootstrap so it should be increased using the /A switch or the modified /FA and /FH switches.

FATAL ERROR (69): ROM image requires a ROM size of  
*number* (hex) bytes

Description This message indicates that the size of ROM specified with the `/R` switch is not big enough to hold the program image.

User Action Increase the size of ROM used and change the `/R` switch to reflect this.

## Chapter 13

# The mempatch Utility

The linker program, `linkt`, normally produces an executable image file prefixed by a short bootstrap program which allows the `afserver` to load the image into an empty transputer: the bootstrap initialises the transputer and reads in the rest of the image file.

The bootstrap produced by the linker is designed to work with the Inmos B004 transputer board, or with an exact copy. These boards have either 1 or 2MB of RAM: the bootstrap may not work properly with partially B004-compatible boards which have different amounts of memory.

This problem does not affect task image files produced by the linker for use with the 3L configurers, since the configurers ignore any bootstrap code prefixed to the input task images and add their own bootstrap to the output application image file. The configurer-generated bootstrap can handle any amount of memory which is a multiple of 64KB.

The linker-generated bootstrap is only used if a single image file is run on its own on one transputer as described in chapter 3. In that case, the following problems may occur on a transputer board with other than 1 or 2MB of RAM:

- On systems with more than 2MB of memory, `.b4` files produced by the linker will assume that only 2MB of memory is available; the program will not be able to take advantage of the rest of the physical memory in the configuration.
- On systems with less than 1MB of memory, `.b4` files produced by the linker will assume too much memory is available, and are likely to fail when memory above the amount actually available is used.
- On systems with more than 1MB but less than 2MB of memory, one or other of the above effects will be observed, depending on the details of the board's address decoding hardware.

The `mempatch` utility allows you to modify `.b4` files so that they will execute correctly with a particular memory configuration other than 1 or 2MB.

The compiler, linker and other utilities provided in this release all use the standard bootstrap, and may therefore require to be modified using `mempatch` if they are to be run on a transputer board with other than 1 or 2MB of RAM. Note that 3L does not guarantee that the compiler, linker and other programs will necessarily operate correctly if insufficient memory is available.

## 13.1 Identifying `mempatch`

If the `mempatch` utility is invoked without arguments, it will print identifying information similar to the following:

```
C>mempatch
usage: mempatch filename.b4 kilobytes
e.g.    mempatch myprog.b4 128
mempatch V1.2, Copyright (C) 1988, 3L Ltd.
```

A given version of `mempatch` can only be guaranteed to operate correctly with particular versions of the 3L high-level languages.



You should only use the version of `mempatch` supplied with this release in conjunction with the corresponding compiler and linker. `mempatch` will detect and reject any program image with which it is not compatible.

## 13.2 Invoking mempatch

The `mempatch` utility is invoked with a command line of the following form:

```
mempatch image-file-name number-of-kilobytes
```

For example, to patch the file `myprog.b4` for a system with only 64 kilobytes of memory, the following command line would be used:

```
C>mempatch myprog.b4 64
standard secondary bootstrap recognised
image now patched to 64 kilobytes
```

Note that the full filename of the program image file—including any `.b4` extension—must be supplied.

## 13.3 Re-invoking mempatch

A program image file may be patched more than once if, for example, available memory in the target system changes. The program file `myprog.b4` modified in the previous example might be modified again for a 128 kilobyte system as follows:

```
C>mempatch myprog.b4 128
previous patch value was 64 kilobytes
image now patched to 128 kilobytes
```



## Chapter 14

# The 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 14.1.

### 14.1 Usage

#### 14.1.1 Compilation for the Decoder

The decoder uses the debugging information generated by the compiler to enable it to produce tables of variable locations and to associate the binary code of the program with the lines of the source file.

For this reason, programs to be decoded should be compiled with the `Zi` switch. For details, see section 9.4.5.

### 14.1.2 Running the Decoder

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 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. Thus, if at compilation time the source file was specified as **down\cats**, and the current directory when the decoder is run is **\mine**, the decoder will attempt to open **\mine\down\cats.c** as the source file. 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. It may, however, be redirected or piped in the usual way, for example:

```
C>decode cats > cats.lis  
C>decode cats | more
```

## 14.2 Features of the decode Program

The line **TOTALCODE 16 0** in the example reports the size of the program code for the module: in this case, 176 (decimal) bytes. The second number can be disregarded.

The line **STATIC 0** in the example reports the size of the static space required by the module. In this case, the module has no static requirements. This value is expressed this time in words (decimal).

Machine-code instructions are decoded into mnemonics. A complete list of all mnemonics produced can be found in appendix E. The decoder automatically merges **pf**ix's and **nf**ix's with the following opcode. There is full support for all T2, T4 and T8 instructions, including the T8's '**fpu**' operations. Unrecognised indirect instructions are decoded as '**opr** *n*', and unrecognised **f**pentry instructions as '**ldc** *n*; **f**pentry'.

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 files added to the program by means of **#include** statement files are not shown, but binary code generated from them is decoded and appears at the right point in the main source file.

At the bottom of the decoder's output we see a decoded representation of the debugging information, including the names of the functions and the displacements of the variables. The size of the debugging information is not included in the **TOTALCODE** and **STATIC** lines above.

### 14.3 Other Languages

The decoder can handle binary object files which are of the format described in the Inmos *Stand-Alone Compiler Implementation Manual*[14]. As well as Parallel C, the 3L Fortran and Pascal compilers generate binary files of this kind, and these can therefore be decoded. If source files are available, the Fortran or Pascal source program will be correctly included in the listing.

```

Transputer DECODE (V1.4) of decodex.bin
ID T8 "occam 2 V2.1" "CC_transputer V2.2.2"
SC 0
TOTALCODE 16 0
STATIC 0

      20 0000F pfix 0
1 void main ()
CODESYMB "main" 00000000
      BE 60 00000 ajw -2
2 {
3   int a, b;
4   a = 100;
      44 26 00002 ldc 100
      D0 00004 stl 0
5   b = a/25;
      70 00005 ldl 0
      49 21 00006 ldc 25
      FC 22 00008 div
      D1 0000A stl 1
6 }
      B2 0000B ajw 2
      F0 22 0000C ret

Debugger Information
Begin Function main 00000000(14) ajw 2 UNKNOWN
SINT32 Wptr 00000000 a
SINT32 Wptr 00000004 b
End Function

```

Figure 14.1: Example of output from `decode`

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 (`.b4`) files.

## Chapter 15

# The worm Utility

The `worm` utility is for exploring transputer networks. In its simplest form, it just counts the number of nodes in the network.

```
C>worm
one processor found
```

The `/F` option switch provides fuller information about each node, including:

- processor type (T414 or T800);
- processor clock speed;
- amount of external memory, in kilobytes (K);
- the number of *extra* processor cycles (penalties) required to access external memory as opposed to on-chip RAM (a minimum of two for a T414 or T800);
- the number of nodes through which work packets in a flood-configured application will be routed to get from the root transputer to this node. This number of “hops” may be greater

than the theoretical minimum imposed by the network configuration; it reflects the network spanning tree constructed by the flood-loading software.

On a single-processor system the output might look like this:

```
C>worm/f
one processor found
processor ROOT type=T414 20.0MHz, 3.0 penalties, 0 hops 2048K
links to HOST[0],-----,-----,-----
```

The link connections from each node are listed from left to right in the order link 0, link 1, link 2, link 3. Here link 0 of the root transputer is connected to the host computer's link adapter and the other three links are unconnected. A dashed line, "-----" indicates an unconnected link.

The `/C` option makes the `worm` generate the node interconnection information in the form of a configuration file suitable for use with the static configurer.

```
! one processor found
processor HOST
processor ROOT
wire ? ROOT[0] HOST[0]
```

## 15.1 Notes

The `worm` will not discover "bare" nodes with little or no external memory. This is because the network loader on which it relies requires about 5–10KB of external RAM to function properly.

There may be a short delay before network information is displayed. This is because the `worm` waits for a certain amount of time before deciding that a link over which nothing is being received is unconnected and not just connected to a "slow" processor.

The `worm` writes its output on the standard output stream, normally the screen. Its output may be redirected to a file, or to a device



like a printer, using the DOS '>' facility. For example, to put a full description of a network into a file called `net.lis`:

```
C>worm/f >net.lis
```



## Chapter 16

# The `tnm` Utility

`tnm` shows the external symbols defined or referenced by an object file or library. For libraries, the names of the constituent object modules are also shown.

`tnm` is invoked like this:

```
tnm filename
```

The *filename* must be the name of an object (`.bin`) file produced by the compiler, or a library file produced by the linker. No default extension is supplied by `tnm`.

Object files and libraries are made up of sequences of object file records of various types. `tnm` scans the input file and writes (to standard output) the following types of record in a printable format. Other record types are skipped.

**COMPILER ID** records show the target processor (**T4** or **T8**) for which a module was compiled, and the version of the compiler used to compile it.

**LIBRARY** records delimit object modules within a library. They also contain the name of the following object module, except for occam

```

LIBRARY MODULE 1: \ims\bin\c\rtlt4\rename.bin
  COMPILER ID occam 2 V2.1 CC_transputer V2.2.2
  REF _put_int
  REF _put_rec
  REF strlen
  REF _get_int
  REF _iob
  CODE SYMBOL rename

LIBRARY MODULE 2: \ims\bin\c\rtlt4\ctable.bin
  COMPILER ID occam 2 V2.1 CC_transputer V2.2.2
  REF _ctype
  DATA SYMBOL _ctype

```

Figure 16.1: `tnm` Output

modules which do not have names and are therefore given numbers instead.

**REF** records name external symbols referred to by the current module. Note that simply referring to a symbol does not cause the module which defines it to be loaded. Only symbols which are actually used in “patch” records cause modules to be loaded. Patch records are not shown by `tnm`, because each symbol may be used in many places in an object file, requiring many patch records which would obscure the output produced.

**CODE SYMBOL** records define the locations of external symbols in the code area of the current module.

**DATA SYMBOL** records define the locations of external symbols in the static data area of the current module.

Figure 16.1 shows the start of the output produced by running `tnm` on the standard C run-time library.

The output from `tnm` normally appears on the screen, but it may be redirected to a file or device using the DOS ‘>’ facility, like this:

```
C>tnm \tc2v2\crtlt4.bin >rtl.lis
```

## Chapter 17

# The `tunlib` Utility

Individual object files can be extracted from a library using the `tunlib` command.

```
tunlib input-library output-library output-objfile symbol
```

All four command-line arguments are required. No default extensions are supplied by `tunlib`.

`tunlib` extracts an object module from the *input-library* and writes it to the *output-objfile*. The *input-library*, minus the extracted module, is copied to the *output-library*.

The module to be extracted is specified by giving the name of any external *symbol* it defines. Symbol matching is case sensitive.

Do not use the same file name as both an input file and an output file. The effects of doing so are undefined.

In the example below, the module which defines the function `PlotPoint` is extracted from a library called `graphlib.bin` and written to an object file of its own called `point.bin`. The remainder of the library is written to a new file, `rest.bin`.

```
C>tunlib graphlib.bin rest.bin point.bin PlotPoint
```

If we had wanted simply to delete the module containing `PlotPoint` from the library, we could have discarded the extracted object file by writing it to the null file, like this:

```
C>tunlib graphlib.bin newlib.bin nul PlotPoint
```

`newlib.bin` is just `graphlib.bin` with the module which defined `PlotPoint` removed.

## Chapter 18

# Configuration Language Reference

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.

### 18.1 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[9]. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory[10].

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.

## 18.2 Configuration Language Syntax

To simplify the explanation of the configuration language, the formal definition which follows in subsections 18.2.2 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 18.2.1.

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 ="`.

### 18.2.1 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.

### 18.2.2 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  $1024 \times 1024$  (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  $1.6 \times 1024 \times 1024$ .

*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	10240
10M	10485760
1.6	1
1.6k	1638

### 18.2.3 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 MS-DOS file names may 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"
"c:\mytasks\x.b4"
"fred.b4"
```

Note that the case of the characters in file names is significant, even though MS-DOS does not use this distinction. This is to help when the software is ported to other environments.

#### 18.2.4 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.

### 18.2.5 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.

### 18.2.6 PROCESSOR Statement

```
processor statement =  
    "PROCESSOR", new identifier, { processor attribute };
```

```

processor attribute =
    "TYPE", "=", processor type |
    "BOOT", "=", boot file specifier |
    "RAM", "=", constant;

processor type =
    "PC";

boot file specifier =
    string constant;

```

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 an IBM PC-type machine). 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 an IBM PC host processor, the host will usually be executing the `afserver` 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 `afserver` 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 IBM PC. 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. For example, a physical network containing one transputer and two IBM PCs might be described as follows:

```
processor host
processor root_processor
processor other_IBM_PC 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]`.

#### 18.2.6.1 BOOT Attribute

The `BOOT` attribute is used to indicate that a processor should not be loaded in the conventional manner but should be booted with the

contents of a named file.

```
processor edge boot="sensor.b4"  
processor gateway boot="anneal.app"
```

At load time a copy of the raw data in the boot file is simply sent to the processor: this can be any code suitable for booting a transputer, including an application image file generated by either the static or flood-fill configurers. In other words, a processor declared with the BOOT attribute can be thought of as the root processor of a sub-network to be booted using the named boot file.

In this way, a main statically-configured network can include static sub-networks or processor farm sub-networks “on the side”. However, these sub-networks must be connected at the *edge* of the main network. There must be only one connection between a sub-network and the main network. If this restriction is not followed, the network may fail to load.

Only the root processor of the sub-network should be described in the main configuration file. If the boot file for the sub-network is a configured application, then a sub-network configuration file will have been used to create it. If the static configurer was used for the sub-network, the sub-network configuration file defines the sub-network topology; this description must be accurate, as no checking can be done during the main network configuration. The processor in the main network which has a BOOT attribute appears in the sub-network configuration file as the **host** processor.

The task in the main network which is to communicate with the root processor in the sub-network must have its ports bound to the appropriate link addresses. The programmer must use the actual hardware addresses for the links to do this. These addresses are as follows:

	Output address	Input address
Link 0	&80000000	&80000010
Link 1	&80000004	&80000014
Link 2	&80000008	&80000018
Link 3	&8000000C	&8000001C

The main task of the sub-network application should be linked with the stand-alone run-time library unless the task it will communicate with in the main network can respond to server protocol (e.g., if the main network task is a file multiplexer).

As an example of a sub-network, if the `upc` application described in chapter 5 were to be split into a main and sub-network using the `BOOT` attribute, the main network configuration file would look like this:

```
! MAINNET.CFG
! Configuration file for upper casing example
! using "boot=" to boot sub-network with upc

processor host
processor root
processor P001 BOOT="subnet.app"

wire ? host[0] root[0]    ! connect PC to transputer
wire ? root[1] P001[2]

! Task declarations
task afserver ins=1 outs=1
task filter ins=2 outs=2 data=10K
task driver ins=3 outs=3

! Assign software tasks to physical processors
place afserver host
place driver root
place filter root

! Set up the connections between the tasks.
connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]

connect ? filter[1] driver[1]
```

```

connect ? driver[1] filter[1]

! bind ports to link to sub-network root processor
bind input  driver[2] value =%80000014    ! I/O over
bind output driver[2] value =%80000004    ! link 1

```

The sub-network configuration file would look like this:

```

! SUBNET.CFG
! Configuration file for uppercasing example. When
! configured this application can be used to boot a
! processor sub-network with the upc program.

processor host          ! really root in main network
processor P001

wire ? host[1] P001[2]

! tasks
task driver ins=3 outs=3
task upc ins=1 outs=1 data=1k

place driver host
place upc P001

connect ? upc[0] driver[2]
connect ? driver[2] upc[0]

```

#### 18.2.6.2 RAM Attribute

The RAM attribute overrides the default mechanism which dynamically determines the amount of memory available to a processor at boot time. The default mechanism probes memory to do this and with certain board designs this is not desirable.

When the RAM attribute is used the configurer will assume that the processor has the amount of memory specified as the parameter to the RAM attribute and the dynamic method of memory determination will not be used. For this reason, care should be taken to ensure that the processor really does have the amount of memory specified with the RAM attribute.

The following RAM attributes declare that processor `pe1` has 4MB of memory and processor `pe2` has only 500KB of memory.

```
processor pe1 ram=4096K
processor pe2 ram=500K
```

Use of the RAM attribute may affect the size of the application file as it may cause extra loading software to be included.

### 18.2.7 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]
```

Although it is only necessary to declare the wires which are actually used by the application, in practice it is advisable to declare all the wires. This is because the configurer may be able to use the extra wires for booting the application, and as a result may be able to reduce the size of the boot file by eliminating some of the loading software.

### 18.2.8 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.

#### 18.2.8.1 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.

### 18.2.8.2 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.

### 18.2.8.3 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 linker program `linkt`.

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 MS-DOS environmental variable `PATH` 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. For example, take the TASK statement `TASK THIS` with no FILE attribute, with the MS-DOS `PATH` variable set up as follows:

```
PATH=c:\mytasks;c:\dos;c:\tputer
```

In this case, the configurer would search for the task image in the following files, in order:

```
.\this.b4
c:\mytasks\this.b4
c:\dos\this.b4
c:\tputer\this.b4
```

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="c:\mytasks\mytask.b4" ...
```

If the identifier-like option is taken, the identifier given is used in a search through the MS-DOS `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 ...
```

#### 18.2.8.4 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.

### 18.2.8.5 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
```

#### 18.2.8.6 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 ...
```

#### 18.2.8.7 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).

### 18.2.9 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 ? afserver[0] filter[0]  
connect ? filter[0] afserver[0]
```

Whereas the WIRE statement describes a hardware connection between links on two transputers, the CONNECT statement describes a logical connection between two tasks. Two kinds of connection are possible:

- A connection between two tasks on the same processor. In this case, the configurer will create a channel word in memory, to which both of the ports will pointed.
- A connection between tasks on adjacent processors, between which there is a free wire.

In the second case, the configurer will map the connection onto a wire. Each wire can support a total of two connections, one in each

direction. If there is no spare wire for the connection, and error message will be output.

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.

### 18.2.10 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 afserver host
```

Where multiple tasks which have the same image file are placed on the same processor, they all share a single instance of the image code. This helps to save space and can be particularly useful for the simulation of large regular systems on fewer processors than will eventually be used.

Note that it is incorrect to PLACE a task on a processor which was declared with a BOOT attribute or on any processor which can only be reached from the host via processors declared with the BOOT attribute.

### 18.2.11 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<sub>16</sub>. 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.

## Chapter 19

# Flood-Fill Configurer Reference

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.

### 19.1 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.

### 19.1.1 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 `afserver` (probably via a protocol filter task such as `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.

### 19.1.2 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.

## 19.2 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 19.1.

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.



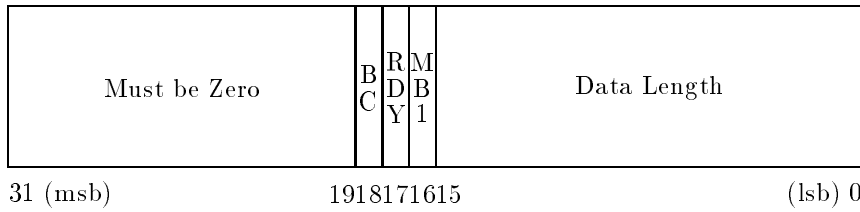


Figure 19.1: Format of Packet Header

- 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.
- Bit number 18 (value  $00040000_{16}$ ) is set to 1 to signify that this packet is a broadcast.
- Bits number 19-31 are always 0.



## Chapter 20

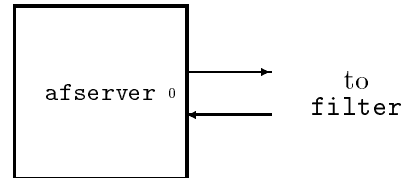
# Task Data Sheets

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: afserver



The **afserver** task is used in configured applications to represent an **afserver** program executing on the host computer. It is therefore not provided in true task-image form.

The **afserver** task should be described to the configurer as follows:

```
task afserver ins=1 outs=1
place afserver host
```

The **afserver** program (and therefore the **afserver** 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 **afserver** is a special variant of the Inmos tagged file-server protocol, adjusted to be tolerant of a problem in the T414A which prevents one-byte messages being sent over links. The **afserver** would therefore normally be attached to a **filter** task so that this variant protocol could be converted back into the protocol which is used by user tasks.

# Data Sheet: filter



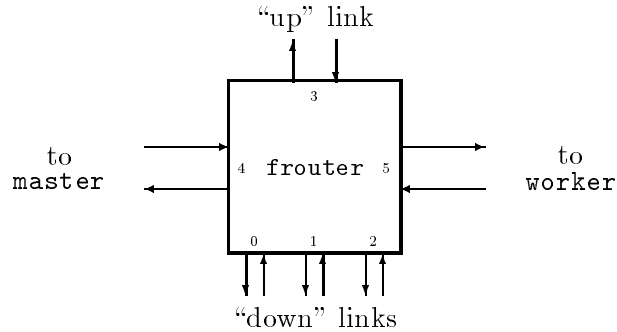
The **filter** task is used to convert between the two extant variants of the Inmos tagged file-server protocol. The two variants arise because of a problem with T414A transputers, which cannot send one-byte messages across links. A **filter** task would be described in a configuration file as follows:

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

A **filter** task's port pair 0 communicates using the T414A-tolerant variant of the Inmos protocol. This is normally attached to an **afserver** task running on the host computer. Port pair 1 of a **filter** task communicates using the standard version of the Inmos protocol.

Thus, if a **filter** task is interposed between an **afserver** 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.

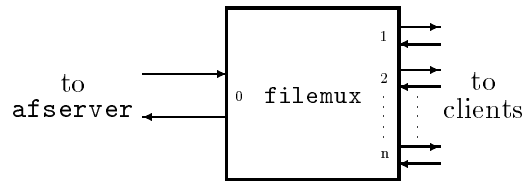
- 4 If this node is the root of the network, these ports are connected to the master task. In this case, the **frouter** task will read work packets and send responses to the master task through this port pair. Otherwise, these ports are set to zero by the loader and the **frouter** task will use port 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 19.1.
- 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.

# Data Sheet: filemux



The **filemux** task allows several client tasks to share a single file server task by merging (multiplexing) the clients' request streams into a single stream of requests. This allows more than one task in a Parallel C application to use standard file I/O. Chapter 6 describes various ways in which this can be done.

In a simple system, the “to **afserver**” ports are connected to the **afserver** via a **filter** task. However, they may be connected to any task which accepts the **afserver** protocol. In particular, they may be connected up as the client of another **filemux** task to build multiplexer chains.

In general, **filemux** simply passes on service requests from its clients and forwards the responses. The exception is the “server terminate” request. The multiplexer will only pass on “server terminate” once all its clients have requested server termination.

Figure 20.1 shows the basic problem with which the multiplexer task is intended to assist. Here, the task **server** runs on the host and provides file services via a protocol filter task **filter** to a client task **client\_1**. The **filter**, **client\_1** and **client\_2** tasks all run in the transputer system. The difficulty is in arranging that the second client task **client\_2** can gain access to files stored on the host processor.



One possibility is to connect the two client tasks together and arrange for `client_2` to request file services from `client_1`. Another possibility, illustrated in figure 20.2, is to introduce a new task **multiplexer** designed to solve this particular problem. The multiplexer task is connected to both client tasks and passes their file service requests through to the filter and thus the server on the host system.

Although it is possible to build any required multiplexing system by combinations of the  $2 \rightarrow 1$  multiplexer shown in figure 20.2, the **filemux** task is more general in that it can handle any number of client tasks: it performs an  $n \rightarrow 1$  multiplexer function. Port pair 0 (i.e., input port 0 and output port 0) of the multiplexer is always connected to the task from which file services may be obtained; in this example, the filter task. All other port pairs supplied to the multiplexer in configuration language statements like `ins= $n$` , `outs= $n$`  are connected to a total of  $n - 1$  client tasks. Any number of client tasks may thus be served by one multiplexer as long as it is provided with sufficient storage to support them all.

An example of a configuration file which represents the configuration of tasks shown in figure 20.2 is given in figure 20.3 (the **processor** and **place** statements required have been omitted for clarity.)

The multiplexer task may also be used to support client tasks which are not running in the root processor. When they are running in

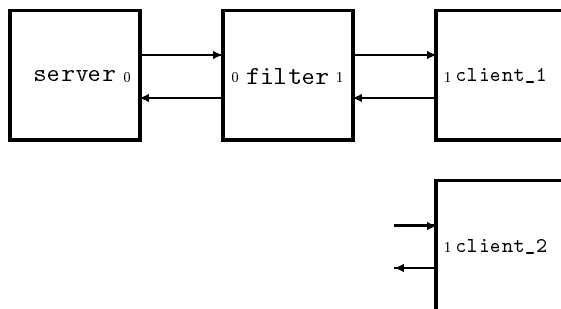


Figure 20.1: Limitation on Server Connections

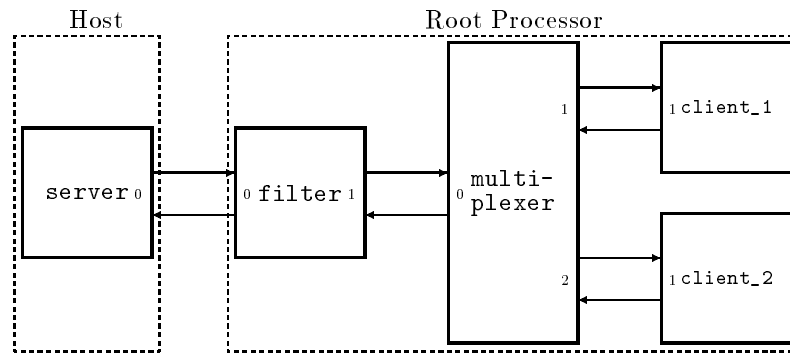


Figure 20.2: Using the Multiplexer

an adjacent processor and there is a spare **wire** connecting the two processors, as in figure 20.4, then no additional work needs to be done; the configurer will simply run the connection between the client and the multiplexer across any available wire. Note that each **wire** between processors, defined in the configuration file, supports bi-directional communication between two tasks, one on each processor.

However, if the client task is some distance away, the multiplexer can be used in a  $1 \rightarrow 1$  configuration (i.e., serving only one client) to pass file service requests through processors in the middle of the network until finally reaching the multiplexer in the root processor, which is connected to the filter task and thus the server as shown in figure 20.5. Thus, a network of transputers might contain a tree of multiplexer tasks, each passing file service requests up towards the root. This kind of arrangement can be continued indefinitely as long as the server task has sufficient resources to handle all the clients together.

As mentioned earlier, the multiplexer can be used in an  $n \rightarrow 1$  manner. An example of its use with eight client tasks (i.e. an  $8 \rightarrow 1$  multiplexer) is shown in figure 20.6. It should be noted that the multiplexer port pair 0 may be connected to one of the client port pairs of another multiplexer task. This allows multiplexers to be chained together to provide file services across a network, if there

```
task server ins=1 outs=1

task filter ins=2 outs=2 data=10k
connect ? filter[0] server[0]
connect ? server[0] filter[0]

task multiplexer file=filemux ins=3 outs=3 data=10k
connect ? filter[1] multiplexer[0]
connect ? multiplexer[0] filter[1]

task client_1 ins=2 outs=2 data=50k

connect ? multiplexer[1] client_1[1]
connect ? client_1[1] multiplexer[1]

task client_2 ins=2 outs=2 data=50k
connect ? multiplexer[2] client_2[1]
connect ? client_2[1] multiplexer[2]
```

Figure 20.3: Example Configuration File

are sufficient links available to do this. Similarly, the `client_8` task might itself be a multiplexer providing file services to tasks on an adjacent processor.

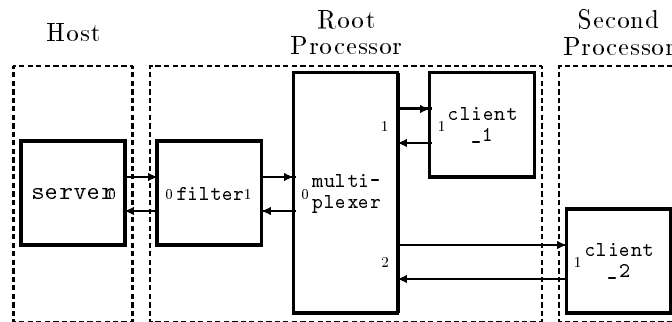


Figure 20.4: Using the Multiplexer on an Adjacent Processor

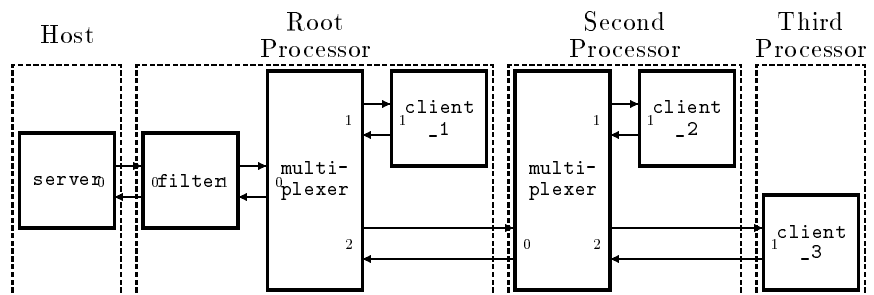


Figure 20.5: Using the Multiplexer from Within a Network

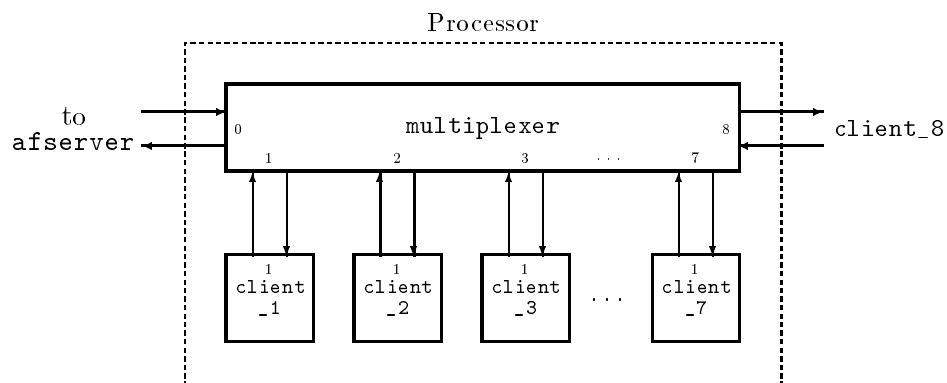
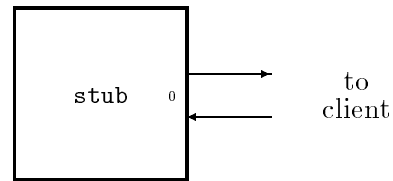


Figure 20.6: Using an  $8 \rightarrow 1$  Multiplexer

# Data Sheet: stub



Tasks which are not connected to the **afserver** or a file multiplexer task are normally linked with the stand-alone C run-time library. There are some standard library functions like **exit** and **sscanf** which do not strictly require file server support but are not in the stand-alone run-time library. The **stub** filer task allows you to write stand-alone tasks which make use of such functions.

All library functions which do not actually require **afserver** support can be made available to a stand-alone task by linking it with the full standard run-time library. If no functions like **printf** are called which require server support, the standard library will only attempt to communicate with the server when it tries to read command-line arguments at program startup and set the exit status at shutdown. The **stub** filer task acts as a sink for these communications: it accepts this limited subset of the **afserver** protocol from its client task and sends back stylised dummy replies.

Note that if the **stub** filer's client task does call a library function which requires server support, e.g., **fwrite**, the **stub** filer will either send back a meaningless response, or terminate and leave the **client** task deadlocked waiting for a response to its request.

The **stub** filer task is connected to its client as shown in the example below. The run-time library always uses output port 1 and input

port 1 to communicate with the server, so the client's port pair 1 must be connected to the **stub** filer's port pair 0.

```
task stub ins=1 outs=1 data=20k
task client ins=3 outs=3

connect ? client[1] stub[0]
connect ? stub[0] client[1]
```

The **stub** filer and its client task act together like an ordinary stand-alone task. In the example above the **client** task has been given three input and three output ports. Port pairs 0 and 1 are reserved for use by the run-time library, so port pair 2 is left free for communication with other tasks.

Use the stand-alone run-time library in preference to the **stub** filer if possible. It is simpler, and the memory used for the **stub** task and some of the startup and shutdown overhead in the full run-time library is saved.

The **stub** filer can only be used with the static configurer, **config**; it cannot be used with the worker task of a flood-filled application. The flood configurer, **fconfig**, will not allow you to specify that a task other than the worker is to be replicated throughout the network.





# Appendix A

## Distribution Kit

This appendix lists the files which are installed on the user's hard disk when the process described in chapter 1 is followed. Each file name is accompanied by a short description of the file's function.

### A.1 Directory \tc2v2

<code>afserver.exe</code>	generic transputer board loader program
<code>t2c.exe</code>	C compiler driver program for T2
<code>t4c.exe</code>	C compiler driver program for T4
<code>t8c.exe</code>	C compiler driver program for T8
<code>tc.exe</code>	generic C compiler driver program
<code>tc.b4</code>	C compiler code for T4 and T8
<code>linkt.b4</code>	linker code
<code>linkt.exe</code>	linker driver program
<code>t2clink.bat</code>	batch file to invoke linker for T2
<code>t4clink.bat</code>	batch file to invoke linker for T4
<code>t8clink.bat</code>	batch file to invoke linker for T8
<code>t4ctask.bat</code>	batch file to link a task for T4
<code>t4cstask.bat</code>	batch file to link a stand-alone task for T4

t8ctask.bat	batch file to link a task for T8
t8cstask.bat	batch file to link a stand-alone task for T8
t4harn.bin	T4 harness code
t8harn.bin	T8 harness code
crtlt4.bin	C run-time library for T4 only
crtlt8.bin	C run-time library for T8 only
sacrtlt2.bin	stand-alone C run-time library for T2
sacrtlt4.bin	stand-alone C run-time library for T4
sacrtlt8.bin	stand-alone C run-time library for T8
alt.h	run-time library header files
ascii.h	
assert.h	
boot.h	
chan.h	
chanio.h	
ctype.h	
dos.h	
errno.h	
float.h	
limits.h	
locale.h	
math.h	
net.h	
par.h	
sema.h	
serv.h	
setjmp.h	
signal.h	
stdarg.h	
stddef.h	
stdio.h	
stdlib.h	
string.h	
thread.h	
time.h	

<code>timer.h</code>	
<code>varargs.h</code>	
<code>decode.b4</code>	code of disassembler utility
<code>decode.exe</code>	<b>decode</b> utility driver program
<code>mempatch.b4</code>	<b>mempatch</b> utility code
<code>mempatch.exe</code>	<b>mempatch</b> utility driver program
<code>tnm.b4</code>	code of object file analyser utility
<code>tnm.exe</code>	<b>tnm</b> utility driver program
<code>tunlib.b4</code>	utility to extract object module from a library
<code>tunlib.exe</code>	<b>tunlib</b> utility driver program
<code>worm.b4</code>	transputer network explorer utility
<code>worm.exe</code>	<b>worm</b> utility driver program
<code>config.exe</code>	configurer driver program
<code>config.b4</code>	configurer code
<code>gloader.b4</code>	loader code used by <b>config</b>
<code>taskharn.t4</code>	harness for tasks on T4
<code>taskharn.t8</code>	harness for tasks on T8
<code>occharn.t4</code>	harness for occam tasks on T4
<code>occharn.t8</code>	harness for occam tasks on T8
<code>fconfig.exe</code>	flood configurer driver program
<code>fconfig.b4</code>	flood configurer
<code>frouter.b4</code>	standard flood router task
<code>floader.b4</code>	loader code used by <b>fconfig</b>
<code>filemux.b4</code>	file server multiplexer task
<code>filter.b4</code>	afserver protocol filter task
<code>stub.b4</code>	stub filer task

## A.2 Directory \tc2v2\examples

<code>hello.c</code>	“hello, world” program
<code>cga.c</code>	source package of functions to access PC’s CGA display hardware from the transputer. Provides an example of use of DOS-access functions.
<code>cga.h</code>	header file for the above

<code>mandelm.c</code>	source of “master” part of Mandelbrot example
<code>mandelw.c</code>	source of “worker” part of Mandelbrot example
<code>mandel.h</code>	header file giving packet formats used by Mandelbrot example
<code>mandel.cfg</code>	configuration file for Mandelbrot example
<code>fmandel.cfg</code>	configuration file for flood-filled version of Mandelbrot example
<code>mandelt4.bat</code>	batch file to compile, link and configure Mandelbrot example for T4
<code>mandelt8.bat</code>	same for T8
<code>driver.c</code>	source of upper-case I/O task
<code>upc.c</code>	source of upper-case conversion task
<code>upc.cfg</code>	configuration file for upper-case example
<code>upct4.bat</code>	batch file to compile, link and configure upper-case example
<code>upct8.bat</code>	same for T8

## Appendix B

# Compatibility with T414A and T800A

This appendix describes the problems which you may encounter if you run Parallel C programs on early transputer chips.

We recommend that if you have one of the development systems sold with these early pre-production processors, you should have it upgraded with a production processor. Failing this, the various problem areas are listed here so that you can program round them.

### B.1 Problems with T414A

Note that the pre-production T414 (mask revision A) *cannot* simply be replaced by a later revision T414 without making changes to the support circuitry. This is because various details of the external clock and phase-locked-loop circuitry differ between the T414A and all later transputer processors. For their own B004 board, Inmos can provide an upgrade kit (IMS B901) which includes a T414B chip, an extraction/insertion tool and full instructions on the modifications required.

### B.1.1 Restriction on Message Lengths

The T414A cannot reliably transmit a single-byte message across a link. Message transfer across internal channels is not affected.

This problem should not affect users of single-transputer systems, as the `filter` task used to communicate with the `afserver` task takes care of this problem. Similarly, the private protocol used between routers in a flood-filled network avoids this problem by padding out 1-byte messages to two bytes for transmission. User tasks in both of these cases are unaware of the protocol conversions.

This problem can be easily avoided in new systems by ensuring that protocols never include single-byte messages.

### B.1.2 Problems with Timers

#### B.1.2.1 Timer Rate Problem

In production transputers, the timer associated with high-priority (“urgent”) threads ticks once every  $1\mu\text{s}$ , while the low-priority timer (that associated with “not urgent” threads) ticks once every  $64\mu\text{s}$ . In the T414A, both timers tick every  $1.6\mu\text{s}$ .

This problem will affect the functions in the `timer` package, those functions in the `chan` package whose names end with `_t`, and the functions in the `boot` package.

#### B.1.2.2 Short Delay Problem

The T414A cannot reliably delay for small amounts of time (below about 5 ticks). When such an operation is attempted, the thread requesting the operation may hang forever.

This problem affects the `timer_wait` and `timer_delay` functions when small delays are specified, and the `thread_deschedule` function, which is equivalent to a 1-tick delay.

## B.2 Problems with T800A

### B.2.1 Floating-Point Conversion Problems

The T800A has a problem in its floating-point microcode; the wrong result may be obtained for expressions containing integer to floating-point conversions.

The Parallel C compiler has an option switch to avoid such instruction sequences; refer to section 9.4.3 for details of the `/T8A` option.

Note that the run-time library supplied with Parallel C has been compiled with this option and can therefore be used safely on a T800A.

### B.2.2 Instruction Decode Problems

The T800A decodes the `move2dzero` and `move2dnonzero` instructions wrongly, with the effect that when one is requested, the other is executed. Later T800 processors decode these instructions correctly, however.

Note that the `/T8A` compiler option does *not* change the behaviour of the assembler with respect to these instructions. The compiler always generates the code for the instruction as written.





# Appendix C

## Building a Network

In order to make use of the multi-processor facilities provided by Parallel C, it is of course necessary to build a multi-transputer network on which to run the programs. This appendix describes the principles involved, and shows how to build such a network out of plug-in transputer development cards for the IBM PC.

### C.1 Network Principles

There are two sets of connections to make when building a network of transputer processors. The most obvious of these are the links connecting one transputer to another; it is through these wires that the tasks running on each processor communicate with their neighbours, and through which the network is bootstrapped. An application running on a transputer network is usually aware of the topology of link connections.

Less obviously, another set of connections must be made in order to arrange that various *system services* are available to the network. Specifically, each transputer processor has *reset* and *analyse* inputs

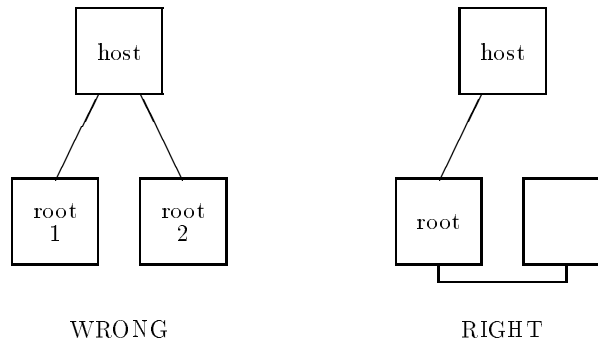


Figure C.1: “One Root” Condition

and an *error* output. The topology of the system service connections need not be related to that of the link connections.

## C.2 Network Requirements

### C.2.1 Requirements for Links

When building a network, there are two conditions which the arrangement of link connections must satisfy:

- Exactly one processor must be connected to the host processor. The former is referred to as the *root* processor, because it forms the root of the structure of processors in the network. Figure C.1 shows two networks, one of which is not acceptable because it attempts to have two root processors.
- Each processor in the network must be reachable by a series of “hops” through links, starting at the host processor. In other words, the network must be *connected*; i.e., have no isolated nodes. Figure C.2 shows two networks, one of which is not acceptable because it has isolated processors.

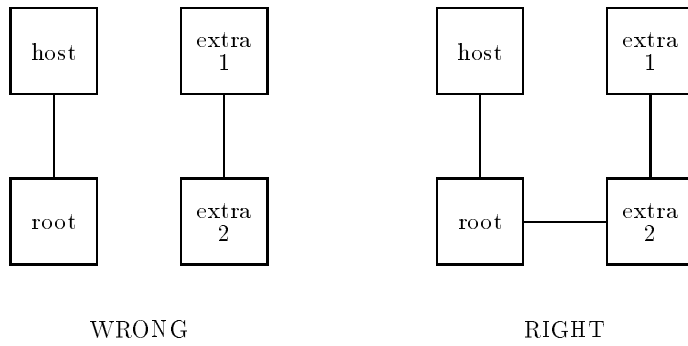


Figure C.2: “Connected” Condition

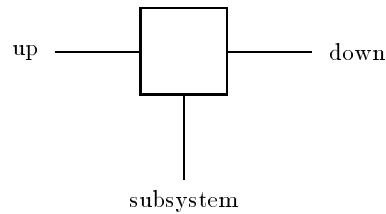


Figure C.3: Inmos System Services Scheme

## C.2.2 Requirements for System Services

The only requirement which Parallel C places on the arrangement of system service connections is that, immediately prior to a network being bootstrapped, all of the processors in that network must have been *reset*. Parallel C makes no use of the transputer analyse and error signals at present.

The reset signal may be carried to each of the processors in the network in many different ways. However, one popular scheme is shown in figure C.3. In this scheme, each processor has three connectors:

- **UP** leads to a processor closer to the host.
- **DOWN** leads to a processor further from the host.

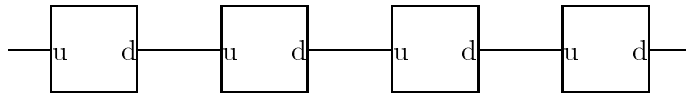


Figure C.4: System Service Daisy Chain

- **SUBSYSTEM** leads to a sub-tree of processors under the control of this one.

The system service signals are carried through from “up” to “down” so that several processors can be “daisy-chained” together. The unconnected “up” port of such a chain can be used to control the entire chain, as shown in figure C.4.

The purpose of the “subsystem” connector is to allow one processor to control others; system service signals are sometimes, but not always, also carried through to the “subsystem” connector.

### C.3 Connecting a Network

This section describes how to connect up a network using boards compatible with the Inmos IMS B004 development board for the IBM PC. The B004 board is shown in figure C.5.

At the far right-hand side of this board, visible from the back of the PC in which the board has been installed, is an array of connectors by which the board may be connected to other boards. There are two columns of five connectors in this array, defined as follows:

PC link	unused
Link 0	Link 1
Link 2	Link 3
PC Reset	Subsystem
Up	Down

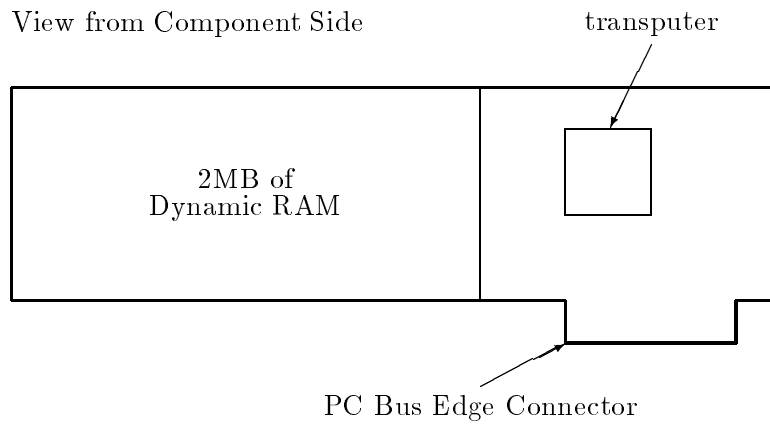


Figure C.5: B004-type Single-transputer Development Board

Boards are supplied with two “jumper” plugs and three cables. These objects are arranged so that they can only fit into the connectors for which they are intended.

When only one development board is in use, the two jumpers are installed. These connect “PC Link” to “Link 0” and “PC Reset” to “Up”; in other words, the board will be reset by the PC in which it is installed, which will load it through its link 0.

To extend this basic configuration with another processor, the second board could be placed in an adjacent PC bus slot (normally to the right) and connections made to carry system services and application messages. For example, link 1 on the root transputer (the original one) could be connected to link 0 on the second board, and “Down” on the root could be connected to “Up” on the add-on. If the two boards are in adjacent slots in the PC card cage, these connectors will be adjacent as well.

This scheme can be extended to any number of development boards; the root (placed on the left) is controlled by the PC, while each board other than the right-most passes the system service signals on to the one on its right.



## Appendix D

# Summary of Option Switches

### D.1 Compiler Switches

Further information can be found in section 9.4, in the subsections specified below for each switch. In the table below, the following notations are used to describe the formats of the switches.

<i>fn</i>	An MS-DOS filename. It may be omitted in whole or in part; the compiler's behaviour in this case is described in section 9.4.
<i>dir</i>	An MS-DOS filename, which will be assumed to refer to a directory.
<i>mac</i>	Any sequence of characters which is acceptable to the compiler as a macro name.
<i>str</i>	Any sequence of characters which is acceptable to the compiler as the value of a macro.
<i>n</i>	A decimal integer.

Switches and their arguments are not case sensitive, except as noted in section 9.4.

<code>/C</code>	9.4.3 Check: do not generate object file.
<code>/Dmac</code>	9.4.7 Define <i>mac</i> with the value 1.
<code>/Dmac=str</code>	9.4.7 Define <i>mac</i> with the value <i>str</i> .
<code>/FBfn</code>	9.4.2 Put binary object output in <i>fn</i> .
<code>/FHfn</code>	9.4.2 Put hexadecimal object output in <i>fn</i> .
<code>/FLfn</code>	9.4.2 Put listing in <i>fn</i> .
<code>/FOfn</code>	9.4.2 Identical to <code>/FB</code> .
<code>/GI</code>	9.4.3 Prevent in-line code generation for library function calls.
<code>/GD</code>	9.4.3 Perform all floating-point arithmetic in double precision.
<code>/GS</code>	9.4.3 Generate 32-bit <b>short</b> variables.
<code>/H</code>	Equivalent to <code>/FH</code> (obsolescent). A <i>fn</i> may not be specified.
<code>/I</code>	9.4.8 Print the compiler's identification.
<code>/Idir</code>	9.4.6 Add <i>dir</i> to the <b>#include</b> list.
<code>/L</code>	Equivalent to <code>/FL</code> (obsolescent). A <i>fn</i> may not be specified.
<code>/M</code>	9.4.8 Include macro expansions in the listing.
<code>/PCn</code>	9.4.4 Set the number of bytes required for an <b>extern</b> function call.
<code>/PMn</code>	9.4.4 Set the number of bytes required for a module number.
<code>/S</code>	9.4.3 Perform floating arithmetic in single precision (ignored).
<code>/T2</code>	9.4.3 Generate object code for the T2 processor.
<code>/T4</code>	9.4.3 Generate object code for the T4 processor.
<code>/T8</code>	9.4.3 Generate object code for the T8 processor.
<code>/T8A</code>	9.4.3 Generate special object code for the Rev A T800 processor.
<code>/Umac</code>	9.4.7 Undefine a predefined macro.
<code>/V</code>	9.4.8 Verbose: display progress messages.
<code>/Wn</code>	9.4.8 Suppress warning messages below severity level <i>n</i> .



/X	9.4.6 Discard the standard <code>#include</code> list.
/ZD	9.4.5 Generate line number tables for <code>decode</code> and debugger.
/ZI	9.4.5 Generate line number tables and debug tables for variables.
/ZO	9.4.5 Generate old format diagnostic information.

## D.2 Linker Switches

The format of the linker's command line and full details of all the switches are discussed in chapter 12. The following is a brief summary of the switches recognised by the linker.

Each switch starts with a slash character `'/'` and an identifying letter; it does not matter if this letter is given in upper case or lower case. The switches can be placed anywhere in the command line but they may not occur in indirect files. No spaces are allowed between a switch's identifying letter and the rest of the switch.

When the *size* of an area is required, you may specify it either as number of bytes (e.g., `4096`) or a number of kilobytes (e.g., `4K`).

/A <i>size</i>	This switch defines the size of stack area
/B <i>file-name</i>	This switch specifies that the file <i>file-name</i> is to be used in preference to the default bootstrap file. There is no default extension for <i>file-name</i> .
/C	This switch stops the linker adding the bootstrap file to the executable file.
/FA	This switch causes the linker to optimise the stack (automatic) area.
/FC	This switch causes the linker to optimise the code area.

/FH	This switch causes the linker to optimise the heap area.
/FS	This switch causes the linker to optimise the static area.
/G	This switch results in the linker creating a debugger information area in the executable or library file.
/I	This switch causes the linker to display its identity and along with various statistics about the executable file such as the code and static sizes and the maximum patch size used.
/L	This switch makes the linker generate a library file rather than an executable file.
/Msize	This switch defines the size of the read-write memory area (including on-chip memory).
/O <i>optimization-symbol</i>	This switch gives priority to the position in the executable image of the object file which defines <i>optimization-symbol</i> .
/O@ <i>optimization-file</i>	This switch gives priority to the position in the executable image of the object files which define the symbols whose names are contained in the file <i>optimization-file</i> . The default extension for <i>optimization-file</i> is <i>.opt</i> .
/P	This switch has the same effect as the /L switch.
/Q	This switch suppresses all warning messages (see section 12.13).
/Q <i>n</i>	This switch suppresses output of message <i>n</i> (see section 12.13).
/Rsize	This switch defines the size of read-only memory area.

- /S** This switch generates a map file taking its name from the first name in the list of object files.
- /Smap-file** This switch generates a map file called *map-file*. The default extension for *map-file* is *.map*.
- /Xentry-point** This switch causes the linker to use the symbol *entry-point* in preference to `INMOS.ENTRY.POINT`, which is the default.

### D.3 afserver Switches

The file server program, **afserver**, is used to load programs from the MS-DOS host into the B004, and to enable programs on the B004 to communicate with the MS-DOS file system and devices. The program should be called like this:

```
afserver command-line redirections
```

where:

*command-line*

is a sequence of switches and program parameters. Anything which is not recognised as a switch is treated as a program parameter. Switches are interpreted by the **afserver**, and not passed to the program. Program parameters are passed to the program, and are ignored by the **afserver**.

*redirections* are used to redirect standard input and output in the usual MS-DOS way. In the case of a Fortran program, standard input and output are preconnected to units 5 and 6 respectively.

For example:

```
C>afserver -:b \tc2v2\tf.b4 /t8 test >errors.lis
```

Here,  `-:b \tc2v2\tf.b4` is an **afserver** switch, and directs it to boot a program, in this case the Fortran compiler. `/t8` and `test` are parameters for the Fortran compiler, and `>errors.lis` redirects the compiler's console output to the file `errors.lis`.

Only **afserver** switches which are relevant to the Parallel C environment are discussed here. Further information may found in the *Stand-Alone Compiler Implementation Manual*[14]. Note that switches may start with  `-:`, as cited below, or  `/:`. Switches must be specified in lower case.

**-:b *file-name***

Boot transputer. The **afserver** will boot the program in *file-name* into the transputer board and start it. Normally, *file-name* will be a `.b4` file output by the linker or one of the configurers. Note that the complete file name must be specified, including the extension.

If a  `-:b` switch is not used, the **afserver** assumes that the transputer board has already been booted, and will try to communicate with the program there.

**-:l *link-address***

Specify link address. By default, the **afserver** uses a block of I/O addresses starting at either  $150_{16}$  or  $300_{16}$  to communicate with the transputer board. It decides which by looking at the host's BIOS *Machine ID*. For all hosts except the original IBM PC,  $150_{16}$  is used. However, the IBM PC uses these I/O addresses for other purposes, and consequently when the machine ID indicates that the host is an IBM PC, the **afserver** uses  $300_{16}$  instead. (There are special varieties of the transputer boards to cope with this.) Unfortunately, the machine ID's of certain IBM-compatible machines (such as the Amstrad PC1512) indicate that they are IBM PCs, even though they more closely resemble the PC/AT. In this case, a  `-:l #150` switch may be used to force the **afserver** to use the correct link address.

A hexadecimal *link-address* is indicated by preceding it with ‘#’.

- :i           Information. The **afserver** prints out its version number, etc.
- :o *flags*    Set program flags. The *flags* are used to set modes for program execution. At present, only two values are recognised.
  - :o 0           The default. Locate the program’s stack on the transputer’s on-chip RAM.
  - :o 1           Locate the whole of the program’s stack in external (off-chip) storage, and use the on-chip RAM for the start of the program code.

More information about these flags may be found in section 3.5.

## D.4 General Purpose Configurer Switches

The General-Purpose Configurer, **config**, is used to assemble a network of tasks into an *application file*. Its operations are controlled by instructions taken from a configuration file, which is written in a special configuration language. This is described fully in chapter 18.

The use of the configurer is described in chapter 5. The program should be called like this:

```
config configuration-file application-file switches
```

where:

*configuration-file*

is the name of the configuration file containing the

instructions for building this application. By convention, configuration files have the extension `.cfg`, but the configurer does not assume this, and the whole file name must be give.

*application-file*

is the name of the application file to be created. Once again, there is no default extension, and the whole name must be supplied.

*switches*      control the configurer's options.

Currently, only one option switch is recognised by the configurer.

*/K*

Normally, the configurer removes debugging information from the task images it loads into the application file. The information is not needed, since Tbug loads a networked application direct from the task-image files, and ignores the application file. Omission of the debugging information also makes the application file smaller.

This switch makes the configurer keep the debugging information. At present we do not recommend using it; it has been implemented now in preparation for future developments in the 3L product range.

## Appendix E

# 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 9.7.

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 the Inmos *Compiler Writer's Guide* [13].

Except for those listed in sections E.5 and E.6 below, all the instructions are available for T2, T4 and T8 transputers.

### E.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.

## E.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 **prefix** and **nprefix** instructions in conjunction 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.

```
prefix ..... prefix
nprefix ..... negative prefix
```

## E.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 **prefix** and **nprefix** 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 T4 and T8 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 **j**, 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)

## E.4 Operations

The instructions in this section are all *indirect* instructions built out of the **opr** instruction. None of these instructions takes 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  
xdbl.....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

```

## E.5 T4-only Instructions

The indirect instructions in this section may only be executed on T4 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

```

## E.6 T8-only Instructions

The instructions in this section may only be executed on T8 processors, although you may use them in **asm** statements even when compiling for a different processor.

### E.6.1 Floating Point Instructions

The indirect instructions in this section provide access to the T8'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 **fentry** instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the **fentry** instruction is also available.

The floating point load and store instructions use the *integer Areg* as a pointer to the operand location.

```

fentry.....floating point unit entry: used to synthesise the
                'fpu...' instructions.
fpdup.....floating duplicate
fprev.....floating reverse
fpldnlsn.....floating load non-local single
fpldnlb.....floating load non-local double
fpldnljni.....floating load non-local indexed single
fpldnljdbi.....floating load non-local indexed double
fpstnljsn.....floating store non-local single
fpstnljdb.....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..... floating-point load zero single  
 fpldzerodb..... floating-point load zero double  
 fpumulby2..... 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

### E.6.2 Other T8-only Instructions

The indirect instructions in this section supplement the T4'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





## Appendix F

# Compatibility Functions

### F.1 Introduction

This appendix describes all those members of the run-time library which are classified as *Compatibility functions*. This means that they are neither defined by the ANSI standard nor supplied by 3L to support the special facilities of the transputer. Functions which fall into one of these groups are dealt with in chapters 10 and 11.

Users should note that none of these functions are recommended for general use. They are non-standard and likely to cause problems with portability, and the run-time library contains equivalent functions with the same or more powerful facilities.

As we saw in chapter 10, declarations of the library functions are held in a number of header files. This applies to the compatibility functions as well, and in the synopsis for each function below, the appropriate header file is indicated by a `#include` statement. In addition, there are three header files which are supplied only for the compatibility functions. They will be considered next.

### F.1.1 ASCII Control Codes <ascii.h>

This header file contains macros defining symbolic names for all the ASCII control codes. The symbolic names defined are those shown in the ASCII code chart in appendix H.

The header file contains no function declarations.

### F.1.2 Channel Communications <chanio.h>

This header file gives access to a group of functions which were used in the earlier versions of Transputer C to perform channel communications. Note that they require the inclusion of `chan.h` as well as `chanio.h`.

<code>_outword</code>	output a word to a channel
<code>_outbyte</code>	output a byte to a channel
<code>_inmess</code>	input a message from a channel
<code>_outmess</code>	output a message to a channel

### F.1.3 Variable Arguments <varargs.h>

This traditional method of accessing variable numbers of arguments has been supplanted in ANSI C by the `stdarg.h` package.

The header defines a type, `va_list`, which can be used to define a pointer to access the arguments. The following macros are defined:

<code>va_alist</code>	used in a function header to specify a variable argument list
<code>va_dcl</code>	declares <code>va_alist</code>
<code>va_start</code>	initialise a pointer to the start of the argument list

`va_arg`        return next argument  
`va_end`        finish accessing arguments

As an example, consider the following function:

```
#include <varargs.h>
int func(va_alist)
va_dcl /* declare variable argument list */
{
    va_list ap; /* pointer to get arguments */
    int arg;
    va_start(ap); /* start getting arguments */
    for (;;) {
        arg = va_arg(ap, int); /* next argument */
        if (arg == 0) break;
        /* process argument */
    }
    va_end(ap); /* stop getting arguments */
}
```

## F.2 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

<b>lseek</b>	places you at a byte offset within a file and returns the new position as an integer
<b>open</b>	opens a file for reading, writing or both
<b>read</b>	reads a specified number of bytes from a file and places them in a buffer
<b>tell</b>	returns the current byte offset within a file
<b>write</b>	writes a number of bytes from a buffer to a file

### F.3 Alphabetic List of Compatibility Functions

The format of the synopses presented here is the same as for those in chapter 11. The boxed indicators have the same meanings.

**\_exit** T2 terminate execution

```
void _exit(int status);
```

**\_exit** closes all the task's files then terminates program execution. It never returns.

**status** is returned to the host operating system as the program result code.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. **exit** should be used instead.

**\_fmode** set default file type

```
extern int _fmode;
```

The variable `_fmode` can be used to change the default behaviour of the `fopen` function. `_fmode` can take two values, which are defined in the file `<stdio.h>` as `O_TEXT` and `O_BINARY`. Note that `_fmode` itself is not declared by `<stdio.h>` and must therefore be explicitly declared as described in the synopsis if it is to be used.

In the default state (`_fmode` equal to `O_TEXT`) a request to `fopen` which does not explicitly request a binary file will result in opening a text file. Some applications written for operating systems in which no distinction is made between text and binary files may have problems with this default, as the effect is to cause expansion of newline characters to the host's local newline convention on output, and to perform a reverse transformation on input.

If the `_fmode` variable is set to `O_BINARY`, the behaviour of `fopen` is changed to opening in binary mode unless text mode is explicitly requested. This means that newline translation to and from the host's local conventions will *not* take place and an application which makes the assumption that text and binary files are not distinguished may be more easily ported to the Parallel C environment.

The use of `_fmode` in this way should be regarded as a last resort for use with applications too large to treat by the preferred method of changing the `type` strings passed to calls on `fopen` to explicitly request binary file access. It is not recommended for use in new applications.

`_inmess` SA T2 read message from channel

```
#include <chan.h>
#include <chanio.h>
void _inmess(CHAN *chanp, char buf[],
             int nbytes);
```

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`. Note that the parameters to `chan_in_message` appear in a different order to those of `_inmess`.

`_outbyte` SA T2 write byte to channel

```
#include <chan.h>
#include <chanio.h>
void _outbyte(char b, CHAN *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` SA T2 write message to channel

```
#include <chan.h>
#include <chanio.h>
void _outmess(CHAN *chanp, char buf[],
              int nbytes);
```

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`. Note that the parameters to `chan_out_message` appear in a different order to those of `_outmess`.

**\_outword** SA T2 write word to channel

```
#include <chan.h>
#include <chanio.h>
void _outword(int w, CHAN *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** MACRO SA T2 convert char to lower case

```
#include <ctype.h>
int _tolower(int cval);
```

If **cval** is the ASCII code for an upper case letter, **\_tolower** returns the code for the corresponding lower case letter. Otherwise, the value of **cval** is returned unchanged.

**\_tolower** behaves like **tolower** but is implemented as a macro.

**\_toupper** MACRO SA T2 convert char to upper case

```
#include <ctype.h>
int _toupper(int cval);
```

If **cval** is the ASCII code for a lower case letter, **\_toupper** returns the code for the corresponding upper case letter. Otherwise, the value of **cval** is returned unchanged.

**\_toupper** behaves like **toupper** but is implemented as a macro.

**cfree** SA T2 deallocates space obtained from heap

```
void cfree(char *ptr);
```

**cfree** has the same function as **free**. It frees the space pointed to by **ptr**, which will have been obtained from the heap by a call to **malloc**, **calloc** or **realloc**.

**close** close a file

```
int close(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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of **close** is **fclose**.

**creat** create a new file

```
int creat(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. The call **creat**(*name*, *mode*) is equivalent to



the call `open(name, 3)`. See the description of `open` on page 469 for details.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of `creat` is `fopen` used with a `type` parameter including a 'w' character.

`ecvt` SA convert floating-point to string

```
char *ecvt(double value, int count, int *dec,
            int *sign);
```

This function is provided for compatibility purposes only, and is not recommended for use in new applications.

`fcvt` SA convert floating-point to string

```
char *fcvt(double value, int count, int *dec,
            int *sign);
```

This function is provided for compatibility purposes only, and is not recommended for use in new applications.

`fdopen` open a stream

```
#include <stdio.h>
FILE *fdopen(int fildes, char *type);
```

`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 260) for a full description of the **type** string.

The **type** of the stream must agree with the way the file was opened.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of **fdopen** is **fopen**.

**fileno** MACRO stream status enquiry

```
#include <stdio.h>
int fileno(FILE *stream);
```

**fileno** returns the low-level I/O “file descriptor” associated with the stream, see **open**. It is implemented as a macro.

**fileno** is used to obtain the low-level file descriptor associated with a high-level stream. The descriptor can be used in calls to the low-level I/O functions (**read**, **write** etc.) when it is desired to mix low-level and high-level operations.

This function is provided for compatibility purposes only, and is not recommended for use in new applications.

**gcvt** SA convert floating-point to string

```
char *gcvt(double value, int digits,
            char *buffer);
```

This function is provided for compatibility purposes only, and is not recommended for use in new applications.

**getw** read an integer from a binary file

```
#include <stdio.h>
int getw(FILE *stream);
```

**getw** reads an integer value from the file referred to by **stream**. The format assumed for the integer is that produced by the **putw** function.

**getw** returns the integer value read; **EOF** is returned if a write error occurs.

**getw** does not assume any particular alignment of the integer value within the file.

**CAREFUL!** The functions **putw** and **getw** should only be used on binary files. If they are used on a text file, the binary data within the integer may be corrupted because of the translation between new-line characters and the host's line termination convention. Note also that the value **EOF** is a valid integer value: if this is placed in a file by **putw** and read out using **getw**, it can be different to determine whether a true integer value is being read or the end of the file has been reached.

This function is provided for compatibility purposes only and is not recommended for use in new applications, where the functions **fwrite** and **fread** can be used to replace **putw** and **getw** respectively.

**index** SA T2 find character in string

```
char *index(char *s, char 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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. It is identical to **strchr** (page 310), which should be used instead.

**isascii** MACRO SA T2 is argument an ASCII character?

```
#include <ctype.h>
int isascii(int cval);
```

Returns  $\neq 0$  if **cval** is an ASCII character (code less than  $80_{16}$ ).

**isatty** is file descriptor a terminal?

```
#include <stdio.h>
int isatty(int fildes);
```

**isatty** returns 1 if the file descriptor **fildes** is associated with the low-level standard input, standard output or standard error files, and 0 otherwise.

**lseek** move read/write pointer

```
long lseek(int fildes, long offset, int whence);
```

The file descriptor refers to an open file. The file position 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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of `lseek` is `fseek`.

**open** open for reading or writing

```
int open(char *name, int mode);
```

According to the `mode` parameter, **open** either creates a new file with the given `name`, or opens an existing file either for reading, writing, for both reading and writing.

`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.

The file is opened in a way determined by the `mode` parameter. This is made up of an access mode (0–3) added to optional file-type flags. The access mode is defined as follows:

- 0 Open the file for reading only.
- 1 Open the file for writing only.
- 2 Open the file for both reading and writing.
- 3 Create the file.

The file-type flags—to be added to the basic access mode—determine whether the file is a text file or a binary file. The flags are defined in `<stdio.h>` as `O_TEXT` and `O_BINARY` respectively. If neither file-type flag is given, the default file-type is taken from the variable `_fmode` (see page 460).

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of `open` is `fopen`.

`putw` write an integer to a binary file

```
#include <stdio.h>
int putw(int ival, FILE *stream);
```

`putw` outputs an integer value to the file referred to by `stream` in a format which can be read in again by the standard input function `getw`.

`putw` returns the integer value written. `EOF` is returned if a write error occurs.

`putw` neither assumes nor causes special alignment in the file.

**CAREFUL!** The functions `putw` and `getw` should only be used on binary files. If they are used on a text file, the binary data within the integer may be corrupted because of the translation between new-line characters and the host's line termination convention. Note also that the value `EOF` is a valid integer value: if this is placed in a file by `putw` and read out using `getw`, it can be difficult to determine whether a true integer value is being read or the end of the file has been reached.

This function is provided for compatibility purposes only and is not recommended for use in new applications, where the functions `fwrite` and `fread` can be used to replace `putw` and `getw` respectively.

`read` read from file

```
int read(int fildes, char *buffer, int 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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of **read** is **fread**.

**rindex** SA T2 find character in string

```
char *rindex(char *s, char 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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. It is identical to **strrchr** (page 314), which should be used instead.

**tell** return file position

```
int tell(int fd);
```

**fd** is a file descriptor returned by **open** or **creat**. **tell** returns the current file position (byte offset) within that file.

If an error occurs, **tell** returns a negative value.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of `tell` is `ftell`.

**unlink** remove a file from the file system

```
int unlink(char *s);
```

This function is identical to `remove`, that is, the file identified by the string parameter `s` is deleted. If the file cannot be removed, the function returns `-1`.

This function is included only for compatibility purposes, and is not recommended for new applications, which should use `remove` instead.

**write** write on a file

```
int write(int fildes, char *buffer, int 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.

This function is provided for compatibility purposes only, and is not recommended for use in new applications. You can use the high-level I/O functions instead of the low-level ones: the high-level equivalent of `write` is `fwrite`.



# Appendix G

## Mandelbrot Program Listings

### G.1 Mandelbrot Example Master Task

```
/** MANDELM.C
**
** Copyright (c) 1988 3L Ltd
**
** Example program: Mandelbrot set evaluation and display.
** NB: This application requires a Colour Graphics Adaptor.
**
** The application
** -----
**
** The application consists of two tasks:
**
** (1) MANDELM (this file). This is the master task, and runs in the
**     root transputer.
** (2) MANDELW. This is the worker task, and runs in all the other
**     transputers of the net.
**
** The flood configurer, FCONFIG, can be used to produce an executable
** file which will automatically distribute the worker tasks across an
** arbitrary network and route work packets from the master to the
** workers.
**
** It is also possible to run the application in a single transputer.
** This will work automatically if the application is configured using
```

```

*** FCONFIG. Alternatively, a static single-transputer configuration
*** could be built by hand, using CONFIG. A suitable configuration file
*** may be found in MANDEL.CFG.
***
*** As well as various routines from the Parallel C run-time library,
*** MANDELM must be linked with the CGA primitives module, CGA.BIN.
*** A file MANDELM.LNK is supplied, which may be used to link MANDELM,
*** like this:
***
***      LINKT @MANDELM.LNK,MANDELM.B4
***
*** Functions of the tasks
*** -----
***
*** MANDELM is told by the user which part of the Mandelbrot set to
*** evaluate. It then breaks this up into 100 packets, and sends them
*** to the network of MANDELW's. As the results from each return, they
*** displayed on the PC's screen.
***
*** Internals of MANDELM
*** -----
***
*** The task contains three threads.
***
*** (1) The MAIN thread.
*** This runs in the function main(). It initialises the other two threads
*** and then goes into a loop, once round for each Mandelbrot display.
*** For each, it gets instructions from the user, and then signals the
*** SEND thread to start work by using the parameters_are_ready semaphore.
*** It keeps track of completed work by examining tally_done, which is
*** incremented by RECEIVE everytime a RESULTS packet is displayed; when-
*** ever it notices that tally_done has changed, it updates the PC's
*** display; and when tally_done reaches 100, MAIN knows that the display
*** is complete.
***
*** (2) The SEND thread.
*** This knows when to start work by examining the parameters_are_ready
*** semaphore. It then breaks the job into 100 small jobs, places the
*** details into a COMMAND structure (defined in file MANDEL.H) and uses
*** the net_send function to send it off to the network of MANDELW's.
*** Notice the SEND does not specify WHICH worker task is to do any
*** particular job; this is decided by the network of router tasks.
***
*** (3) The RECEIVE thread.
*** This simply waits till a packet arrives from the network of MANDELW's
*** and then displays it. Each packet contains all the necessary
*** information to display it, so RECEIVE does not need to keep track of
*** which packet is which. Every time it does a display, RECEIVE
*** increments tally_done, so that MAIN can tell when the whole display
*** is complete.
***
*** Rev 001 21-Jul-89 ADC make send(), receive() 'void' instead of default

```

```

***          'int' to match thread_create() prototype for its
***          first (function pointer) parameter.
***  Rev 000 16-Dec-87 JF
***
***/

#include <stdio.h>
#include <dos.h>
#include <thread.h>
#include <sema.h>
#include <par.h>
#include <net.h>
#include "cga.h"
#include "mandel.h"

/* Interface to SEND thread */
static SEMA parameters_are_ready;

/* Interface to RECEIVE thread */
static int tally_done;

/* Current Mandelbrot and display parameters */
static float x_coord, y_coord, gap;
static int thresh1, thresh2, thresh3;

/* Define the way the job is broken into packets */
#define X_INCREMENT ((CGA_LORES_XMAX+1)/10)
#define Y_INCREMENT ((CGA_YMAX+1)/10)
#define PACKETS 100

/*
 * This function is invoked by MAIN using thread_create to
 * create the SEND thread.
 */

void send ()
{
    int          x, y;
    COMMAND      c;

    for (;;) {

        /* Wait here until MAIN signals it's okay to go ahead */
        sema_wait (&parameters_are_ready);

        /* Fill in the fixed parts of the command */
        c.x_coord = x_coord;
        c.y_coord = y_coord;
        c.gap = gap;
    }
}

```

```

/* Send off the packets to be done. Each includes the coordinates
of the top-left and bottom-right corners of the area to do.
This both tells the worker task what values to generate and
identifies the RESULTS packet when it arrives in the RECEIVE
thread (since there's no guarantee that the results will arrive
in the same order the commands are sent out) */
for (x = 0; x < CGA_LORES_XMAX; x += X_INCREMENT) {
    c.tlx = x; c.brxc = x + X_INCREMENT - 1;
    for (y = 0; y <= CGA_YMAX; y += Y_INCREMENT) {
        c.tly = y; c.bry = y + Y_INCREMENT - 1;
        /* Send off the next packet */
        net_send (sizeof(COMMAND), &c, 1);
    }
}

}

}

/*
 * This function is invoked by MAIN using thread_create to
 * create the RECEIVE thread.
 *
 */

void receive ()
{
    RESULTS      r;
    int          len, ready, x, y, i, n, colour;

    for (;;) {

        /* Thread will wait here till a packet arrives */
        len=net_receive (&r, &ready);
        i = 0;

        /* The results packet includes the coordinates of the top-left
        and bottom-right corners of the data, so we know where to
        display it. */
        for (y=r.tly; y<=r.bry; y++) {
            for (x=r.tlx; x<=r.brxc; x++) {
                n = r.counts[i++];
                /* Received 0 means 1; received 255 means 256 */
                n += 1;
                /* Decide on the colour <- thresholds, and display... */
                colour = (n>=thresh1) + (n>=thresh2) + (n>=thresh3);
                cga_lores_plot (x, y, colour);
            }
        }

        /* Increment the tally of packets displayed */
        tally_done += 1;
    }
}

```

```

    }
}

/*
 * The MAIN thread runs here
 *
 */

main ()
{

    float range;
    int previous_tally;

    /* Make sure we have text mode (and clear screen), then sign on */
    video_mode (MONO_80COL_TEXT_MODE);
    printf ("\nCopyright (c) 1988 3L Ltd\n\n");
    printf ("Example program: Mandelbrot set evaluation and display\n");
    printf ("NB: This program requires a Colour Graphics Adaptor\n\n");

    /* Initialise this SEMA to 0 BEFORE we start the SEND thread.
       This means it will wait until we tell it it's safe to go ahead */
    sema_init (&parameters_are_ready, 0);
    /* Now start the other two threads */
    thread_create (send, 10000, 2,0,0);
    thread_create (receive, 10000, 2,0,0);

    for (;;) {

        /* This will ensure that no other threads are using the C
           run-time library (in fact, in this case they won't be,
           but I have done it here as an example...) */
        sema_wait (&par_sema);
        printf ("\nInput X coordinate: "); scanf ("%f", &x_coord);
        printf ("Input Y coordinate: "); scanf ("%f", &y_coord);
        printf ("Input Y range:      "); scanf ("%f", &range);
        gap = range / (float)(CGA_YMAX+1);
        y_coord = y_coord + range;

        printf ("Threshold 1: "); scanf ("%d", &thresh1);
        printf ("Threshold 2: "); scanf ("%d", &thresh2);
        printf ("Threshold 3: "); scanf ("%d", &thresh3);
        getchar (); /* Consume the final NL */

        /* We have finished with the C RTL - release it */
        sema_signal (&par_sema);

        /* Into graphics (CGA low resolution) mode */
        video_mode (CGA_LORES_GRAPHICS_MODE);
    }
}

```

```

/* Before we set SEND going, reset the count of finished
   packets to zero - RECEIVE will count it back up          */
tally_done = 0;
/* All ready - set it going! */
sema_signal (&parameters_are_ready);

previous_tally = 0;

/* Until all the packets have been done, just keep updating the
   display when necessary                                     */
while (tally_done < PACKETS) {
    while (tally_done==previous_tally) {
        /* Wait here till something happens. Use thread_deschedule
           to save cpu time                                     */
        thread_deschedule ();
    }
    /* Send the picture up to the PC's display memory */
    cga_update ();
    previous_tally = tally_done;
}

/* In case tally_done was updated to = PACKETS AFTER the last
   cga_update, do another one to ensure the PC's display is
   up-to-date                                                */
cga_update ();

/* One again, wait for the RTL to be free; then beep and wait
   till the user strikes any key                             */
sema_wait(&par_sema);
putchar ('\007');
getchar ();
sema_signal(&par_sema);

/* Clear the screen and set text mode again */
video_mode (MONO_80COL_TEXT_MODE);
}
}

```

## G.2 Mandelbrot Example Worker Task

```

/** MANDELW.C
**
** Copyright (c) 1988 3L Ltd
**
** Example program: Mandelbrot set evaluation and display.
** NB: This application requires a Colour Graphics Adaptor.
**
** The application
** -----
**
** The application consists of two tasks:
**
** (1) MANDELM. This is the master task, and runs in the root
**     transputer.
** (2) MANDELW (this file). This is the worker task, and runs in all the
**     other transputers of the net. It uses the 'net_' library functions
**     to obtain work packets originating from MANDELM and send back
**     result packets.
**
** A file MANDELW.LNK is supplied, which may be used to link MANDELW,
** like this:
**
**     LINKT @MANDELW.LNK,MANDELW.B4
**
** For further details, see the top of MANDELM.C.
**
** Internals of MANDELW
** -----
**
** The task waits till a packet arrives. This is a COMMAND struct,
** containing details of the portion of the Mandelbrot to do. It
** then does the work, storing the results in a RESULTS struct, which
** is then sent back to MANDELM.
**
** */

#include <net.h>
#include "mandel.h"

static COMMAND c;
static RESULTS r;

main ()
{
    int    x, y, count, n, ready;
    float  gap, x_coord, y_coord,
           ac, bc, two=2.0, four=4.0, size, a2, b2, a, b;

```

```

for (;;) {

    /* Task will wait here until a packet arrives */
    n = net_receive (&c, &ready);
    /* Unpack some of the parameters */
    x_coord=c.x_coord;
    y_coord=c.y_coord;
    gap=c.gap;

    /* The top-left and bottom-right coordinates are supplied
       in the command packet */
    n = 0;
    for (y=c.tly; y<=c.bry; y++) {
        bc = y_coord - y*gap;
        for (x=c.tlx; x<=c.brx; x++) {
            ac = x*gap + x_coord;
            a = ac; b = bc; size = 0.0; count = 0;
            /* Do calculation until more than 2.0 away or
               until count reaches 256 */
            a2 = a*a; b2= b*b;
            while ((size < four) && (count < 256)) {
                b = two*a*b + bc;
                a = a2 - b2 + ac;
                a2 = a*a; b2= b*b;
                size = a2 + b2;
                count++;
            }
            /* Stored 0 means 1; stored 255 means 256 */
            r.counts[n++] = count-1;
        }
    }

    /* Send the top-left and bottom-right coordinates back in the
       RESULTS packet too, so that the RECEIVE thread of MANDELM can
       identify the packet. */
    r.tlx = c.tlx; r.tly = c.tly;
    r.brx = c.brx; r.bry = c.bry;
    net_send (16+n, &r, 1);

}

}

```



## G.3 Header File

```

/**  MANDELTY.H
**
**  Parallel Mandelbrots
**
**  These are the formats of the packets used to communicate between
**  the master task and the computation tasks.
**
**  Rev 000   6-Dec-87   JF   Created
**
**/

typedef struct command_structure {
    float  x_coord, y_coord, gap;
    int    tlx, tly, brx, bry;
} COMMAND;

typedef struct results_structure {
    int    tlx, tly, brx, bry;
    char   counts[1008];
} RESULTS;

```

## G.4 Configuration File

```

Processor Host
Processor Root

Wire ? Host[0] Root[0]

Task Afserver Ins=1 Outs=1
Task Filter   Ins=2 Outs=2 Data=10K
Task MandelM  Ins=2 Outs=2 Data=500K
Task MandelW  Ins=1 Outs=1 Stack=1K Heap=10K Opt=Stack Opt=Code

Connect ? Afserver[0] Filter[0]
Connect ? Filter[0]   Afserver[0]
Connect ? Filter[1]   MandelM[1]
Connect ? MandelM[1]  Filter[1]
Connect ? MandelM[0]  MandelW[0]
Connect ? MandelW[0]  MandelM[0]

Place Afserver Host
Place Filter   Root
Place MandelM  Root
Place MandelW  Root

```



# Appendix H

## ASCII Code Chart

	0x0x	0x1x	0x2x	0x3x	0x4x	0x5x	0x6x	0x7x
0xx0	NUL	DLE	␣	0	@	P	‘	p
0xx1	SOH	DC1	!	1	A	Q	a	q
0xx2	STX	DC2	"	2	B	R	b	r
0xx3	ETX	DC3	#	3	C	S	c	s
0xx4	EOT	DC4	\$	4	D	T	d	t
0xx5	ENQ	NAK	%	5	E	U	e	u
0xx6	ACK	SYN	&	6	F	V	f	v
0xx7	BEL	ETB	'	7	G	W	g	w
0xx8	BS	CAN	(	8	H	X	h	x
0xx9	HT	EM	)	9	I	Y	i	y
0xxA	LF	SUB	*	:	J	Z	j	z
0xxB	VT	ESC	+	;	K	[	k	{
0xxC	FF	FS	,	<	L	\	l	
0xxD	CR	GS	-	=	M	]	m	}
0xxE	SO	RS	.	>	N	^	n	~
0xxF	SI	US	/	?	O	_	o	DEL



# Bibliography

- [1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, First Edition*. Prentice-Hall, 1978. ISBN 0-13-110163-3.
- [2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988. ISBN 0-13-110362-8.
- [3] *American National Standard for Information Systems - Programming Language - C*. American National Standards Institute, Inc, 1990. X3.159-1989.
- [4] *Disk Operating System Version 3.10 Reference*. International Business Machines, February 1985.
- [5] *Microsoft MS-DOS User's Reference*. Microsoft Corporation, 1986. Document Number 410630013-320-R03-0686.
- [6] *Disk Operating System Version 3.00 Technical Reference*. International Business Machines, May 1984.
- [7] A. M. Lister, *Fundamentals of Operating Systems*. Macmillan Press, 1979. ISBN 0-333-27287-0.
- [8] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*. Prentice-Hall, 1987. ISBN 0-13-637331-3.

- [9] *British Standard BS6154:1982: Method of Defining Syntactic Metalanguage*. British Standards Institution, 1981. ISBN 0-580-12530-0.
- [10] R. S. Scowen. *An Introduction and Handbook for the Standard Syntactic Metalanguage*. National Physical Laboratory Report DITC 19/83, February 1983.
- [11] *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, 1985.
- [12] Inmos Ltd. *Transputer Reference Manual*. Prentice-Hall, 1988. ISBN 0-13-929001-X.
- [13] Inmos Ltd. *Transputer Instruction Set: A compiler writer's guide*. Prentice-Hall, 1988. ISBN 0-13-929100-8.
- [14] *Stand alone compiler implementation manual*. Version 1.1, Inmos Ltd., July 1987.
- [15] *TDS Compiler implementation manual*. Version 1.0, Inmos Ltd., November 19, 1986.

# Index

- '<', *see* I/O redirection
- '>', *see* I/O redirection
- '|', *see* I/O redirection
- '@', *see* linking: with indirect files
- '%', *see* object files: hexadecimal
- '-', *see* **afserver**: options
- '\$', *see* identifiers: dollar sign in
- .b4, *see* executable files,
  - task image files,
  - application image files
- .bin, *see* object files
- .c, *see* source files
- .cfg, *see* configuration files
- .dat, *see* linker: and indirect files
- .lib, *see* linker: creating library files
- .lis, *see* compiler: listing files
- .map, *see* linker: and map files
- .opt, *see* linker: and optimisation files
- /
- /C, 125, 128
- /D, 132
- /F, 124–126
- /FB, 125
- /FL, 125–126
- /F0, 125
- /Gd, 127
- /Gi, 127
- /Gs, 128
- /I, 132–133, 136
- /M, 133
- /P, 129–130
- /PC, 129
- /PM, 130
- /S, 126
- /T2, 126
- /T4, 126
- /T8, 126
- /T8A, 126
- /U, 133, 135
- /V, 134
- /W, 135
- /X, 132, 136
- /Zd, 131
- /Zi, 131
- /Zo, 131
- 
- \_3L\_SHORT\_BITS, 128
- \_exit, 460
- \_filer\_handle, 238
- \_fmode, 226, 460
- \_inmess, 458, 461
- \_outbyte, 458, 462
- \_outmess, 458, 462
- \_outword, 458, 463
- \_tolower, 463
- \_toupper, 463
- 3**
- 3LCC\_INC, 4, 136
- A**
- abort, 231, 239
- abs, 232, 239
- acos, 218, 239
- afserver, 19
  - command-line parameters, 21

- invoking, 19, 46, 443
- limit on open files, 71
- switches, 19, 21, 25, 46, 443
- task data sheet, 412
- version, 8
- `alloc86`, 215–216, 239
- `alt_nowait`, 209, 240
- `alt_nowait_vec`, 209, 241
- `alt_wait`, 209, 241
- `alt_wait_vec`, 209, 242
- application files, *see* application image files
- applications, 30
- `argc`, 22, 121, 205
- `argv`, 22, 121, 205
- ASCII, 483
- `asin`, 218, 243
- assembler, 137
  - error messages, 198
  - labels and jumps, 144–145
  - literal bytes, 146, 448
  - opcodes, 447
  - operands, 139–142
  - syntax, 138
  - uses for, 137
- `assert`, 209, 243
- `atan`, 218, 244
- `atan2`, 218, 244
- `atexit`, 231, 244
- `atof`, 230, 244
- `atoi`, 230, 245
- `atol`, 230, 245
- `autoexec.bat`, 3–4

## B

- batch files
  - for linker, 13, 16, 348–349
  - for running, 20
- `bdos`, 214, 216, 246
- binary files, *see* object files
- BIND statement, 405
- `boot_peek`, 209, 246
- `boot_poke`, 209, 247
- bootstrap
  - and the linker, 346

- configurer, 367
  - for T2, 99
  - standard, 367–368
- bootstraps, 346
- broadcasts, *see under* processor farms
- `bsearch`, 232, 247
- `BUFSIZ`, 305
- byte, 147
- `BYTE_REGS`, 213

## C

- `calloc`, 231, 248
- `ceil`, 219, 248
- `cfree`, 464
- CHAN, 49
- `chan_in_byte`, 211, 249
- `chan_in_byte_t`, 211, 249
- `chan_in_message`, 211, 250
- `chan_in_message_t`, 211, 250
- `chan_in_word`, 211, 250
- `chan_in_word_t`, 211, 251
- `chan_init`, 211, 249
- `chan_out_byte`, 211, 251
- `chan_out_byte_t`, 211, 251
- `chan_out_message`, 211, 252
- `chan_out_message_t`, 211, 252
- `chan_out_word`, 211, 252
- `chan_out_word_t`, 211, 253
- `chan_reset`, 211, 253
- channels, 27–29, 209
- `clearerr`, 230, 254
- `clock`, 235, 254
- `close`, 459, 464
- command-line parameters, *see main*, `afserver`
- compiler, 89
  - and floating-point, 127
  - and processor types, 126
  - bit fields, 120, 150
  - code gaps, 129
  - controlling verbosity, 134
  - debug tables, 131
  - default switches, 123
  - differences from K&R C, 110



- disassembling output from, 371
  - error message lists, 157, 193, 196, 198
  - error messages, 151–154, 156
  - file defaults, 124
  - identifying, 133
  - inlining functions, 127
  - invoking, 12, 121
  - list of keywords, 111
  - listing files, 125, 133
  - module numbers, 130
  - object files, 125
  - options, 122
  - output files, 124
  - representation of data types, 147
  - shifts, 120
  - size of external call, 129
  - size of module numbers, 130
  - special features, 119
  - switch summary, 439
  - switches, 122
  - temporary files, 122
  - version, 8
    - see also* `#include`, macros,
  - configuration files, 34, 38, 45, 77, 83
    - more than one transputer, 53
  - configuration language
    - anonymous identifiers, 390
    - file layout, 385
    - identifiers, 389
    - link specifiers, 393
    - numeric constants, 387
    - port specifiers, 402
    - statement syntax, 391
    - string constants, 388
    - syntax of, 383
  - configurer, 33–34, 37–38
    - and T2, 103
    - debug tables, 446
    - invoking, 44–45, 445
    - switches, 445
    - see also* flood-fill configurer,
  - CONNECT statement, 43, 403
  - connections between ports
    - declaring to configurer, 403
  - `const`, 111
  - conventions
    - filename extensions, 14, 16, 18
  - `cos`, 218, 254
  - `cosh`, 218, 254
  - `creat`, 459, 464
- ## D
- data, *see* compiler: representation of data types
  - debug tables, *see under* compiler, linker, configurer
  - debugging
    - parallel systems, 63, 220
    - see also* errors,
  - decode, 131
  - `decode`, 371
    - invoking, 372
  - disassembly, 371
  - distribution kit
    - contents, 425
    - installing, 1
    - testing, 7
  - `div`, 232, 255
  - DOS, *see* MS-DOS
  - `dos.h`, 214
- ## E
- `ecvt`, 465
  - `entry`, 111
  - `enum`, 111
  - environmental variables
    - `3LCC_INC`, 4, 136
    - `TC`, 124
    - `PATH`, 399
    - `TMP`, 122
  - `EOF`, 206
  - `errno`, 218, 255
  - error messages, *see under* compiler, linker
    - redirecting, 13
  - errors
    - bizarre, 8, 20, 25, 394
    - linker, 350–351, 353–365

- patch over valid code, 129
- program hangs, 25
- EventReq**, 210
- example programs
  - “hello, world”, 12
  - Mandelbrot, 77, 80, 83–84, 473
  - MS-DOS access, 215–216
  - multiplexer, 54–55
  - upper case, 31, 39–40
- executable files, 13, 335, 338–340
  - as MS-DOS commands, 20
  - created by linker, 16
  - rules for inferring name of, 335
  - running with **afserver**, 19
- execution, *see* running
- exit**, 231, 256
- exp**, 218, 256

## F

- fabs**, 219, 257
- fclose**, 227, 257
- fcvt**, 465
- fdopen**, 465
- feof**, 230, 257
- ferror**, 230, 257
- fflush**, 227, 258
- fgetc**, 228, 258
- fgetpos**, 229, 258
- fgets**, 228, 259
- FILE**, 223
- filemux**, 67–76
  - memory requirements, 69
  - task data sheet, 416
- fileno**, 466
- filter**, 38
  - task data sheet, 413
- filters, *see* I/O redirection
- floating point
  - IEEE, 148
- floating-point
  - constants, 114, 127
  - evaluation of expressions, 113, 127
  - format, 148
  - infinity, 148, 265

- Not-a-number, 148, 265
- flood-fill configurer, 80, 83, 407
  - heterogenous networks, 86
  - invoking, 84
  - task-task protocol, 407
  - see also* processor farms,
- floor**, 219, 259
- fmod**, 219, 260
- fopen**, 227, 260
- fprintf**, 227, 261
- fputc**, 228, 266
- fputs**, 228, 266
- fread**, 229, 267
- free**, 231, 267
- free86**, 215, 217, 268
- freopen**, 227, 268
- frexp**, 219, 268
- from86**, 217, 269
- frouter**, 80
  - task data sheet, 414
- fscanf**, 227, 269
- fseek**, 229, 273
- fsetpos**, 229, 274
- ftell**, 229, 274
- fwrite**, 229, 274

## G

- gcvt**, 466
- general-purpose configurer, *see* configurer
- getc**, 228, 275
- getchar**, 228, 275
- getenv**, 231, 276
- gets**, 229, 276
- getw**, 467
- global I/O, 67–76
  - application termination, 72
  - see also* filemux,

## H

- hardware
  - assumptions, xvi, 367
  - configuration, 34, 40
  - troubleshooting, 8

harness  
     standard, 17, 38–39  
     T4 and T8 versions, 17  
     task, 38, 44

heap storage, *see under* memory

host processor, 40  
     special treatment of, 46

**I**

I/O  
     global, *see* global I/O  
     redirection, *see* I/O redirection

I/O redirection, 13, 22–23

identifiers  
     case distinction, 112  
     dollar sign in, 120  
     in configuration language, 40, 43  
     reserved as keywords, 111  
     significant characters, 111

#include  
     controlling, 132, 136  
     directory search, 135

index, 467

indirect files, *see under* linker

infinity, *see under* floating point

INMOS.ENTRY.POINT, 348

inp, 217, 277

installation  
     directory, 1, 3

int86, 214, 216, 277

int86x, 214, 216, 278

intdos, 214, 216, 278

intdosx, 214, 216, 279

interrupts, *see under* MS-DOS

isalnum, 212, 279

isalpha, 212, 279

isascii, 468

isatty, 459, 468

iscntrl, 212, 279

isdigit, 212, 280

isgraph, 212, 280

islower, 212, 280

isprint, 212, 280

ispunct, 212, 281

isspace, 212, 281

isupper, 212, 281

isxdigit, 212, 281

**L**

labs, 232, 282

ldexp, 219, 282

ldiv, 232, 282

library files, 17, 336–338  
     changing, 17  
     compared to indirect files, 337  
     creating, 17–18, 337–338, 347  
     debug information in, 338  
     extracting members, 381–382  
     inferring the name of, 337  
     listing contents, 379–380  
     using, 336

Link0Input, 210

Link0Output, 210

Link1Input, 210

Link1Output, 210

Link2Input, 210

Link2Output, 210

Link3Input, 210

Link3Output, 210

linker, 13–15, 17, 333, 350–351,  
     353–363, 365  
     and bootstraps, 346  
     and debug tables, 338, 345, 347  
     and indirect files, 15, 18,  
         335–336  
     and library files, 336  
     and map files, 348  
     and optimization files, 348  
     and patching gaps, 129–130  
     batch files for, 13, 16  
     command line, 333–334  
     creating library files, 17  
     duplicate definitions, 349  
     entry points, 348  
     error messages, 350–351,  
         353–363, 365  
     file name conventions, 334–335  
     invoking, 16  
     libraries, *see* library files  
     map files, 340

- messages, 350
- modified */F* switches, 344
- more than one object file, 14
- optimization files, 339–340
- optimization symbols, 338–340, 347
- ordering of object files, 334
- patch over valid code, 129
- simple programs, 13
- supports only 1MB or 2MB, 25
- switches, 18, 345–348, 441–443
- T2 support, 93–97, 340
- version number, 347
- linking
  - stand-alone tasks, 52
  - tasks, 44
- links, 29, 393
- linkt, 16
- listing files, *see under* compilers
- localeconv, 217, 283
- log, 219, 283
- log10, 219, 283
- longjmp, 221, 284
- lseek, 460, 468

## M

- macros
  - defining, 132
  - listing expansions, 133
  - pre-defined, 91
  - predefined, 132–135
- main, 22, 30, 121, 204
  - on T2, 106
- malloc, 231, 284
- master, 83
- master task, 77–79, 408
  - see also* processor farms,
- mblen, 232, 285
- mbstowcs, 232, 285
- mbtowc, 232, 286
- memchr, 234, 286
- memcmp, 233, 287
- memcpy, 233, 288
- memmove, 233, 288
- memory, 23

- code storage, 23
- estimating requirements, 65–66
- external, 24–25
- heap storage, 24
- limits imposed by linker, 25
- on-chip, 21, 24, 338
- physical, 24
- run-time library requirements, 23
- speed of, 25
- stack, 23
- static storage, 23
- storage areas, 23
- mempatch, 25, 367–368
  - compatibility, 368
  - identifying, 368
  - invoking, 369
- memset, 234, 288
- messages, 27–28, 350–365
  - length of, 48
- modf, 219, 289
- MS-DOS
  - accessing functions of, 212
  - filters, *see* I/O redirection, 23
  - search path, 3, 45, 399
  - versus PC-DOS, xviii

## N

- NaN, *see under* floating point
- NDEBUG, 243
- net package, 81
  - buffer sizes, 81
  - multiple packets, 81
- net\_broadcast, 82, 220, 289
- NET\_MAX\_PACKET\_LENGTH, 81
- net\_receive, 78–81, 220, 290
- net\_send, 78–79, 81, 220, 290
- Not-a-Number, *see under* floating point
- NotProcess\_P, 210
- NUL, 238, 483
- NULL, 206, 208, 238, 291

## O

- object files, 13, 16–17

- format of, 16
- ordering of in executable file, 334, 338–340
- offsetof**, 208, 292
- on-chip memory, *see* memory:
  - on-chip
- open**, 460, 469
- options, *see* compiler:switches,
  - linker:switches,
  - afserver**:switches,
  - configurer:switches
- outp**, 217, 292

**P**

- par\_fprintf**, 220, 293
- par\_free**, 220, 292
- par\_malloc**, 220, 293
- par\_printf**, 220, 293
- par\_sema**, 220, 294
- PC-DOS, *see* MS-DOS
- pcpointer**, 214
- perror**, 230, 294
- pipes, *see* I/O redirection
- PLACE statement, 42, 404
- port vectors, 31
- portability, 112
- ports, 30, 48–49, 402
  - binding, 31, 405
- pow**, 219, 295
- printf**, 228, 295
- processes, 27–28
- processor farms, 34–35, 77, 81–82, 289
  - and broadcasts, 82, 289
  - networking software, 80–81
  - routing software, 289
  - see also* master task, worker
    - task, **frouter**, flood-fill
    - configurer,
- PROCESSOR statement, 40, 391
  - BOOT attribute, 393
  - RAM attribute, 396
  - TYPE attribute, 393
- processor type
  - compiling for, 13, 126

- differing on-chip memory, 24
- harnesses for, 17
- linking for, 14, 16
- run-time libraries for, 17
- T2, 89–106
- T414A, 38, 415, 429
- T800A, 126, 429, 431
- processors
  - declaring to configurer, 392
- program parameters, *see* **main**,
  - afserver**
- ptrdiff\_t**, 208
- putc**, 229, 295
- putchar**, 229, 296
- puts**, 229, 296
- putw**, 470

## Q

- qsort**, 232, 297

## R

- raise**, 222, 298
- rand**, 230, 299
- read**, 460, 470
- realloc**, 231, 299
- redirection, *see* I/O redirection
- register**, 121
- REGS, 213
- remove**, 227, 300
- rename**, 227, 300
- rewind**, 229, 300
- rindex**, 471
- root transputer, 40
- run-time library, 13, 17
  - and **afserver**, 19
  - and processor types, 204
  - binary I/O, 226
  - channel I/O functions, 209
  - character testing functions, 211
  - compatibility functions, 457
  - conventions, 205
  - DOS functions, 212
  - full, 204
  - header files, 206

- heap functions, 231
- input/output, 223
- list of functions, 238–261, 266–269, 273–316, 318–331, 460–472
- mathematical functions, 217
- memory requirements, 23
- network functions, 220
- parallel I/O functions, 220
- purpose, 203
- semaphore functions, 221
- stand-alone, 51–52, 204
- stream I/O, 225
- string handling functions, 233
- T2 version, 99
- T4 and T8 versions, 17
- text I/O, 226
- thread functions, 235
- time functions, 235
- timer functions, 236
- variable arguments, 222
- running, 19–23, 44
  - off-chip stack, 25
  - on T2, 102
  - on-chip stack, 24

## S

- `scanf`, 228, 301
- scheduling
  - see also* priority,
- search path, *see under* MS-DOS
- `segread`, 214, 216, 301
- `sema_init`, 221, 301
- `sema_signal`, 221, 302
- `sema_signal_n`, 221, 302
- `sema_test_wait`, 221, 303
- `sema_wait`, 221, 303
- `sema_wait_n`, 221, 304
- semaphores, 33, 60, 80, 221
- `serv_filter`, 221, 304
- server, *see* `afserver`
- `setbuf`, 227, 305
- `setjmp`, 221, 306
- `setlocale`, 217, 306
- `setvbuf`, 227, 307

- short integer variables, 112, 128, 147
- `signal`, 222, 307
- `sin`, 218, 308
- `sinh`, 218, 308
- `size_t`, 208
- `sizeof`, 119
- source files
  - conversion from TDS, 12
  - creating, 12
- `sprintf`, 228, 308
- `sqrt`, 219, 309
- `srand`, 230, 309
- `sscanf`, 228, 309
- stack, *see under* memory
- stand-alone library, *see* run-time library: stand-alone
- standard error, 224
- standard input, 22, 224
- standard output, 22, 224
- static storage, *see under* memory
- `stderr`, 224
- `stdin`, 22, 224
- `stdout`, 22, 224
- storage, *see* memory
- `strcat`, 233, 310
- `strchr`, 234, 310
- `strcmp`, 233, 310
- `strcoll`, 233, 311
- `strcpy`, 233, 311
- `strcspn`, 234, 311
- `strerror`, 234, 312
- `strlen`, 234, 312
- `strncat`, 233, 312
- `strncmp`, 234, 313
- `strncpy`, 233, 313
- `strpbrk`, 234, 313
- `strrchr`, 234, 314
- `strspn`, 234, 314
- `strstr`, 234, 315
- `strtod`, 230, 315
- `strtok`, 234, 316
- `strtol`, 230, 316
- `strtoul`, 230, 318
- `strxfrm`, 234, 318

**stub**  
     task data sheet, 422  
 switches, *see under* compiler, linker,  
     **afserver**, configurer  
 system, 231, 319

## T

T2, *see* processor type  
 T4, *see* processor type  
 T414A, *see under* processor type  
 t4clink, 13–14  
 t4cstask, 52  
 t4ctask, 44  
 t4master, 86  
 t4worker, 86  
 T8, *see* processor type  
 T800A, *see under* processor type  
 t8clink, 14  
 t8cstask, 52  
 t8ctask, 44  
 t8master, 86  
 t8worker, 86  
 tan, 218, 319  
 tanh, 218, 320  
 task data sheets, 411  
     **afserver** task, 412  
     **filemux** task, 416  
     **filter** task, 413  
     **frouter** task, 414  
     **stub** task, 422  
 task files, *see* task image files  
 task image files, 30, 34, 45  
     locating, 45  
     locating with configurer, 399  
 TASK statement, 41, 397  
     FILE attribute, 83, 86, 399  
     INS attribute, 41, 398  
     memory size attributes, 400  
     OPT attribute, 401  
     OUTS attribute, 41, 49, 399  
     URGENT attribute, 402  
 taskharn.t4, 45  
 taskharn.t8, 45  
 tasks, 30–32  
     declaring to configurer, 398

    normal versus stand-alone, 51  
     specifying memory  
         requirements, 400  
     versus threads, 60  
     *see also* task image files, TASK  
         statement,  
 Tbug, 63, 131, 345, 347  
 TC, 124  
 TDS, 4, 12  
 tdslist, 12  
 tell, 460, 471  
 temporary files, 122  
 thread\_create, 235, 320  
 thread\_deschedule, 235, 321, 431  
 THREAD\_NOTURG, 235  
 thread\_priority, 235, 321  
 thread\_restart, 235, 322  
 thread\_start, 235, 322  
 thread\_stop, 235, 323  
 THREAD\_URGENT, 235  
 threads, 33, 57, 80  
     creating, 57, 60  
     versus tasks, 60  
 time, 235, 323  
 timer\_after, 236, 324  
 timer\_delay, 236, 324, 431  
 timer\_now, 236, 325  
 timer\_wait, 236, 325, 431  
 timers, *see under* transputer  
 tmpfile, 227, 325  
 tmpnam, 227, 326  
 tnm, 379–380  
 to86, 217, 327  
 tolower, 212, 326  
 toupper, 212, 327  
 transputer  
     byte, 147  
     channels, 209  
     error flag, 20  
     links, 210  
     on-chip RAM, *see* memory:  
         on-chip  
     timers, 236  
     word, 147

*see also* channels, links,  
processor type,  
*see also* processor type,  
tunlib, 381–382

## U

ungetc, 229, 327  
unlink, 472  
unsigned, 150  
unsigned short, 147  
URGENT, 402

## V

va\_alist, 458  
va\_arg, 222, 328, 459  
va\_dcl, 458  
va\_end, 222, 328, 459  
va\_start, 222, 329, 458  
variables  
    stack, 23  
    static, 23  
vfprintf, 228, 329  
void, 111  
volatile, 111  
vprintf, 228, 330  
vsprintf, 228, 330

## W

wchar\_t, 208  
wcstombs, 232, 331  
wctomb, 232, 331  
WIRE statement, 40, 397  
wires, 29  
    declaring to configurer, 397  
word, 147  
WORDREGS, 213  
work packets, 77–79, 408  
worker, 83  
worker task, 77–79, 408  
    *see also* processor farms,  
workspace, *see* memory: stack  
worm, 375  
write, 460, 472