ANSI C toolset user manual

INMOS Limited

72 TDS 224 00

August 1990

INMOS is a member of the SGS-THOMSON Microelectronics Group.

The C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler.

UNIX is a trademark of AT&T.

INMOS document number: 72 TDS 224 00

Contents overview

Tools

Appendices

Contents

Preface

About this Manual

This manual is a User Guide to the ANSI C toolset. The manual is divided into two main parts, plus appendices:

1 **User Guide**. Describes the toolset and shows how it is used to develop and run transputer programs.

2 **Tools**. Detailed descriptions of the individual tools, with their syntax and options.

3 **Appendices**. For technical reference.

Differences from previous toolsets

Differences from the 3L Parallel C toolset are listed immediately after this preface.

Host versions

The manual is designed to cover all host versions of the toolset:

IMS D7214 – IBM and NEC PC running MS-DOS.
IMS D5214 – Sun 3 systems running SunOS
IMS D4214 – Sun 4 systems running SunOS
IMS D6214 – VAX systems running VMS

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
`Teletype`	Used to distinguish command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote an optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis ...	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

Differences from 3L Parallel C

This chapter lists the differences between ANSI C and the previous 3L Parallel C toolset.

List of differences

1 The ANSI C compiler implements ANSI standard C. The 3L Parallel C compiler was an implementation of basic K & R C.

2 The ANSI C compiler is invoked by the **icc** command, which replaces the **tc** series of commands in 3L Parallel C. Transputer targets are now specified using command line options.

The compiler is completely new and command line options may have different meanings.

3 The ANSI C toolset makes use of the new TCOFF object file format. This means that object files created with 3L Parallel C are not compatible with object files created using ANSI C. If possible 3L source should be recompiled. If this is not possible then the file conversion tool **icvlink** can be used to convert 3L object files to the new TCOFF format.

4 3L Parallel C supported T4 and T8 processor types. ANSI C compiles code for all currently supported transputer types.

5 The linker **ilink** is completely new and command line options may have different meanings.

6 The harness and the runtime library, previously required on the linker command line are not required in the ANSI C toolset and are replaced by the linker indirect file **startup.lnk** which references all the runtime and library code required.

7 The default extensions for the binary object file output from the compiler and linker are **.tco** and **.lku** respectively; in Parallel C they were **.bin** and **.c*xx***.

Although the filename conventions used in 3L Parallel C can still be used a new set exists for the ANSI C toolset. See sections 2.11 and A.4.

8 The 3L configurer tool **config** is now defunct and is replaced by **icconf**.

No equivalent to the 3L configurer tool **fconfig** exists in the ANSI C toolset.

9 The configuration language is completely new.

10 The **iboot** tool is now defunct and is replaced by **icollect**. **icollect** generates bootable files for single and multi-transputer programs from single linked units and configuration binary files respectively.

11 The **decode** utility is not supplied with the ANSI C toolset. The binary lister tool **ilist** provides equivalent functionality.

12 The librarian **ilibr** is completely new and command line options may have different meanings.

13 Tools have been added for creating ROM-based programs. **ieprom** formats bootable code for installing into EPROMs and **iemit** assists in creating memory configurations. The conversion tool **icvemit** is provided for converting memory configurations created by the earlier **iemi** tool.

14 A comprehensive debugger **idebug** is provided which supports source level debugging, low level debugging, and breakpointing.

15 An **imakef** tool is provided to assist with program building.

16 A transputer simulator tool **isim** is provided to run and test programs without transputer hardware.

17 The ANSI C Runtime Library is an implementation of the ANSI standard library plus some INMOS specific extensions. Many extra functions have been added that were not present in 3L Parallel C and a new concurrency support library is provided.

Most of the functions present in 3l Parallel C are also represented in ANSI C. Where functions have been omitted it is because they are either no longer required or there exists an equivalent ANSI C function.

3L functions *not* included in ANSI C are:

_inmess	**_outmess**	**_outbyte**	**_outword**
_tolower	**_toupper**	**boot_poke**	**boot_peek**
fdopen	**fileno**	**index**	**net_receive**
net_send	**putw**	**rindex**	**serv_filter**
cfree	**getw**		

Functions in the 3L packages **thread**, **sema**, **timer**, **chan**, and **par** are retained for compatibility but all the functions are now declared in the header file **conndx11.h**. The functions now simply call equivalent functions in the new concurrency library and may operate slower than if the equivalent ANSI C functions were called directly.

18 ANSI C, like Parallel C, provides a reduced version of the Runtime Library for modules which do not communicate with the host. This library is installed in the file **libcred.lib** and can be linked to a program by specifying the linker indirect file **startrd.lnk** on the linker command line.

19 Mixed language programming can be achieved in the ANSI C toolset by configuring linked units created using TCOFF toolsets on any processor. Facilities are provided for calling occam from C.

20 In line assembly code is now introduced with the keyword **__asm**. The transputer code facility is extended with additional syntax.

Comparison of commands

This section shows a comparison of the commands required to generate and run a program on a T8 series transputer. For simplicity in presentation the examples are given using the '-' option switch character only.

3L Parallel C:

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

ANSI C:

```
icc f1 -t8
icc f2 -t8
ilink f1.tco f2.tco -f startup.lnk -t8 -o
main.lku
icollect main.lku -t
iserver -sb main.btl
```

User Guide

72 TDS 224 00

August 1990

1 Introduction to transputers

This chapter introduces transputers and concurrent programming. It describes how the transputer supports concurrent programming through on-chip hardware and introduces the concepts of parallel processing in C.

1.1 Overview

Parallel processing is a powerful way of increasing system performance and can be applied whatever the underlying architecture. The combination of hardware concurrency support and a compiler toolset which makes the hardware features easily accessible from software makes the transputer and toolset a powerful vehicle for the development of parallel applications.

1.2 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware. They can be connected together by their serial links in application-specific ways and can be used as the building blocks for complex parallel processing systems.

The transputer is a complete microcomputer on a single chip. In addition to the hardware support for processor communications it contains a very fast (single cycle) on-chip memory and a programmable memory interface that allows external memory to be added with the minimum of supporting logic.

Figure 1.1 shows the generalised architecture of the IMS T4 family of 32 bit transputers.

1.2.1 Multitransputer systems

Multitransputer systems can be built very simply. The four high speed links allow transputers to be connected to each other in arrays, trees, and many other configurations. The circuitry to drive the links is all on the transputer chip and only two wires are needed to connect two transputers together.

Some possible arrangements of transputers are illustrated in figure 1.2.

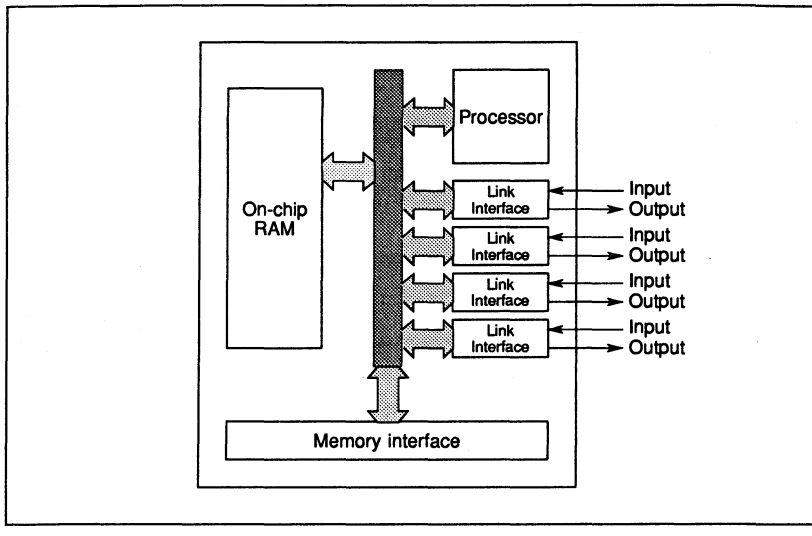


Figure 1.1 Transputer architecture

1.2.2 Links

In addition to providing a communication and synchronisation path between processors, transputer links allow memory to be examined directly by debugging programs and permit programs to be loaded onto whole networks of transputers down a single transputer link. Each individual transputer also supports communication between parallel processes through a system of internal links implemented as words in memory.

1.2.3 Hardware parallel support

Each transputer has a highly efficient built-in run-time scheduler for processes running in parallel on the same transputer and supports channel communication through single words in memory. Processes waiting for input or output, or waiting on a timer, consume no CPU resources, and process context switching time can be as little as one microsecond. The communication links between processors operate concurrently with the processing unit and can transfer data simultaneously on all links without the intervention of the CPU.

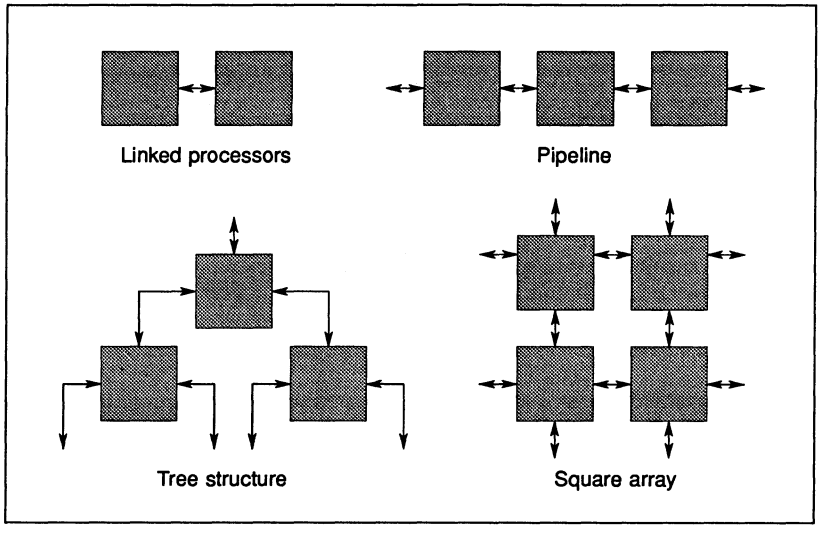


Figure 1.2 Transputer networks

1.2.4 Transputer products

There is a complete family of transputer devices, including: 32 bit and 16 bit processors; a peripheral control processor; a link switch; and a parallel link adaptor.

A wide range of transputer programming boards is supplied by INMOS and other vendors for several hosts. These boards can be used for:

- Developing and debugging transputer software
- Improving system performance (as accelerator boards)
- Loading software onto embedded systems
- Building specific transputer networks.

1.3 Transputers and C

The ANSI C toolset has been designed to reflect the parallel processing model of communicating sequential processes (CSP). The inherent flexibility of the C language, the capacity to mix code from different languages, and the ability to

use the concurrency features of the transputer make ANSI C a powerful tool for programming concurrent systems.

1.3.1 Programming model

The parallel programming model consists of a number of independent processes executing simultaneously and communicating through **channels**. Channels are one-way communication paths that allow processes to exchange data.

A process can be built from any number of other parallel processes, so that an entire software system can be described as a hierarchy of intercommunicating parallel processes. This model is consistent with many modern software design methods.

Communication between processes is synchronised. When data is passed between two processes the output process does not proceed until the input process is ready. Buffered communication and multiplexing can be achieved by inserting a specific buffer or multiplexing process between the two processes. Library functions are provided for the input and output of data on channels.

1.3.2 Multitransputer programming

Processes are independently executable and will run on any processor in a network. A special configuration language is used to distribute processes over a network of transputers and can be used to program complex multiprocessor systems.

1.3.3 Real time programming

The concurrency features of the transputer provide direct support for real time programming. The key features are listed below.

- Direct and efficient implementation of parallel processes in hardware
- Prioritisation of parallel processes.
- The ability to implement software interrupts as high priority processes
- Easy programming of software timers, allowing close control of timing and non-busy polling
- Placement of variables at specific addresses in memory, for accessing memory mapped devices.

Some of the technical issues in transputer programming are discussed in the INMOS series of Technical Notes. Selected titles in this series are listed in the bibliography towards the rear of this manual.

1.4 Program development

The compiler and its supporting tools run under standard operating systems, either on the host itself or on a transputer board attached to the host, and can be used in conjunction with existing text editing software and source control systems. For this reason, no editor is provided with the toolset.

1.4.1 Software design

The software designer can use ANSI C to specify the components of a system in terms of communicating processes. The overall design can be directly expressed in the parallel constructs of the language.

Common modules can be collected together into libraries for the purpose of code sharing within programming teams.

1.4.2 Programming

Code for single transputers is linked using the linker tool and loadable programs are generated using the collector tool. For multitransputer systems the collector tool reads and processes a configuration data file created by the configurer tool; for single transputer programs the collector adds bootstrap code for a single processor. Single processor bootstrapping by the collector is controlled by a command line option.

Software processes and channels are allocated using the configuration language and loadable code ready for distribution on the network is generated using the configurer.

1.4.3 Debugging

Programs for multi-processor systems can be debugged at the symbolic level using the network debugger that allows a breakpointed or halted program to be analysed in terms of its source code. A low level debugging environment using direct memory display, instruction disassembly, and processor data is also provided. Breakpoint debugging allows programs to executed interactively, and post-mortem debugging allows stopped programs to be debugged from the contents of the transputers' memory. The debugger inserts no additional code into

the program, but rather reads data from a description file. This guarantees that the code generated when debugging is disabled will always run in the same way as the final version of the program.

1.4.4 Embedded systems

Programs for embedded systems can be loaded from the host directly onto the target hardware via a transputer link. If the program is to be held in ROM, special tools are provided to reformat the object code for loading into an EPROM or for processing by user-defined EPROM loader programs. A configuration tool is provided to assist with the evaluation and definition of specific memory configurations.

2 Overview of the toolset

This chapter gives an overview of the ANSI C toolset. It briefly describes each tool, outlines its purpose, and explains how the tools are used together to develop, configure, load and run transputer programs. The chapter also introduces the runtime library, outlines the standards for error reporting, and summarises host-specific characteristics.

2.1 Introduction

The ANSI C toolset is a software cross-development system for transputers, hosted on PC/MS-DOS, Sun 3/SunOS, Sun 4/SunOS and VAX/VMS systems. It consists of a full ANSI C compiler with concurrency support, a multilanguage linker, a configurer for mapping programs onto transputer networks, a code collector tool for generating directly loadable files, and a program loader and host server tool. The toolset also includes a fully interactive debugger, program build tools, and EPROM programming tools. Together, the compiler and its supporting tools form an integrated environment for the development of programs on transputers and transputer-based hardware.

2.2 Features of the toolset

The ANSI C toolset is an integrated development system for transputer programs incorporating a new standard object file format, a C-like configuration language, a comprehensive Runtime Library, and support for concurrent programming based on the communicating process model. It represents a broad enhancement of the approach to parallel programming in C and introduces standards for the generation of object code for transputers and transputer-based hardware.

2.2.1 Standard object file format

The ANSI C compiler generates object code in an intermediate form known as **TCOFF** (*T*ransputer *C*ommon *O*bject *F*ile *F*ormat). The adoption of a common format introduces a standard for the development of future transputer compilers and enables code generated by compatible compilers to be freely mixed in the same system.

2.2.2 New configuration language

The toolset incorporates a new configuration language that allows software and hardware networks to be described separately and joined by a software-to-

hardware description. The language is a simple declarative language that has the syntactic flavour of C and can be used on any size of network. A full range of high level language constructs including replicative and conditional statements make it easy to explore different configurations before committing to hardware.

2.2.3 Runtime library

A comprehensive runtime library is supplied with the toolset providing full ANSI C support with additional support for concurrency and parallel programming. The library of concurrency functions provides the choice of either channel-based or semaphore-based communication. An optimised library with no server support is available for embedded code.

2.2.4 Concurrent programming

The abstract model used in ANSI C reflects the Communicating Sequential Process (CSP) model of parallel programming. The model maps easily onto the transputer to provide efficient parallel code. Software is broken down into independent processes which exchange data and synchronize their activity via channels. Processes can be mapped onto one, several, or many transputers using the new configuration language.

2.2.5 Transputer targets

The ANSI C toolset can be used used to write programs targetted at IMS M212, T212, T222, T225, T400, T414, T425, T800, T801, and T805 transputers. Code can also be written to run on a group of processor types by compiling for a transputer *class*.

2.2.6 Support for earlier toolsets

A file convertor tool supplied with the toolset enables object code and libraries generated by earlier INMOS compilers and toolsets such as the 3L Parallel C and occam 2 toolsets to be incorporated into programs written with ANSI C. Specific support is provided for functions from the 3L Parallel C toolset.

2.3 Toolset summary

The tools provided in the toolset are summarised in Table 2.1 and briefly described in the following sections.

Tool	Description
`icc`	The ANSI C compiler. A full ANSI standard compiler with concurrency support. Generates object code for specific transputer targets.
`icconf`	The configurer. Analyses the configuration description and produces a configuration data file for the code collector.
`icollect`	The code collector. Collects linked units into a single file for loading on a transputer network. Takes as input a configuration data file or a single linked unit.
`icvlink`	The TCOFF file convertor. Converts object files generated by earlier toolsets to TCOFF format.
`idebug`	The network debugger. Provides post-mortem and interactive debugging of transputer programs.
`idump`	The memory dumper. A debugging auxiliary tool used to debug programs that run on the root transputer.
`iemit`	The transputer memory configuration tool. Used for evaluating and defining memory configurations for later incorporation into ROM programs.
`ieprom`	The EPROM program formatter tool. Formats transputer bootable code for input to ROM programmers.
`ilibr`	The toolset librarian. Builds libraries of compiled code in the same format as the C runtime library.
`ilink`	The toolset linker. Resolves external references and links separately compiled code into a single file.
`ilist`	The binary lister. Disassembles and decodes object code and displays information in a readable form.
`imakef`	The Makefile generator. Generates Makefiles for input to MAKE programs.
`iserver`	The host file server. Loads programs onto transputer hardware and provides runtime access to the host.
`isim`	The T425 simulator. Simulates program execution on an IMS T425 transputer and provides simple debugging facilities.
`iskip`	The skip loader tool. Used with `iserver` to load programs onto external networks over the root transputer.

Table 2.1 The ANSI C toolset

2.4 ANSI C compiler – `icc`

The compiler `icc` is an ANSI standard C compiler with additional support for concurrency. It conforms fully with ANSI standard X3.159 1989.

The ANSI standard for C formalises the original implementation of C as described in '*The C Programming Language*' by Kernighan and Ritchie, and extends it to include a runtime library, some language extensions already in common usage, and many other improvements designed to standardise the language.

The original implementation of C will be referred to in the rest of this manual as 'K&R C' and ANSI standard C as 'ANSI C'. A summary of the differences between K&R C and the ANSI standard can be found in section .1.

ANSI C supports concurrency through a series of C structures and a comprehensive set of process handling, channel communication, and semaphore manipulation functions. Some useful non-ANSI functions are also provided in the runtime library.

The compiler produces compiled code for specific processor types or transputer classes. The compiled object file is in a standard intermediate code format which must be linked, configured, and made executable before the program can be run. The runnable file consists of code which can be directly loaded onto a transputer network.

2.5 Generating executable code

Three tools are used in sequence (or two for a single transputer program) to generate the loadable file from compiled object code:

`ilink` – the toolset linker which links separately compiled program units

`icconf` – the configurer tool which generates a configuration data file (multitransputer programs only)

`icollect` – the code collector which generates a bootable file for a transputer network either from the configuration data file or a single linked unit.

2.5.1 Linker – `ilink`

The toolset linker **`ilink`** links separately compiled modules and libraries into a single code unit, resolving external references and generating a *linked unit*. Linked units can be used in configuration descriptions to map software onto spe-

cific arrangements of transputers, or can be bootstrapped for a single transputer using **icollect**.

Library modules are linked in with the program by the the C startup file which must be specified on the linker command line. The correct startup file must be specified for the transputer type.

2.5.2 Configurer – icconf

The configurer **icconf** generates configuration information for transputer networks from a *configuration description* written in the transputer configuration language. The tool prepares the program for configuring on a specific arrangement of transputers by analysing the configuration description and producing a data file for the code collector tool.

2.5.3 Code collector – icollect

The code collector tool **icollect** takes the data file generated by **icconf** and generates a single file that can be loaded and run on a transputer network. The file contains bootable code modules for all processors on the network along with distribution information that is used by the loader to place the modules on each processor.

icollect is also used to generate bootable code for single transputer programs from linked units by appending single transputer bootstrap code. The single transputer mode of operation is selected by a command line option.

2.6 Loading and running programs

Bootable code for single transputers and transputer networks is loaded onto the transputer hardware using the host file server tool **iserver** which both loads the program and starts up the runtime environment that supports interaction with the host. The auxiliary skip loading tool **iskip** can be used in combination with **iserver** to load a program onto an external network.

2.6.1 Host file server – iserver

The host file server **iserver** is a combined host server and loader tool. When invoked to load a program it both loads the code onto the transputer hardware and provides runtime services on the host (such as program i/o) for the transputer program.

2.6.2 Skip loader – `iskip`

The skip loader **iskip** forces a program to be loaded over the root transputer (the transputer connected to the host). It is used prior to invoking **iserver** for loading programs onto a transputer board without needing to use the root transputer as part of the network. The tool is useful when debugging programs that are configured to use the root transputer because it leaves the root transputer free to run the debugger and avoids the use of **idump** to save the program image.

2.7 Program development and support

Seven tools are provided to assist in program development:

idebug – the interactive network debugger.

idump – the memory dump tool for use with **idebug** when debugging programs on the root transputer.

ilibr – the librarian which generates libraries of compiled code.

ilist – the binary lister which decodes and displays data from object files.

isim – the T425 transputer simulator.

imakef – the Makefile generator which creates Makefiles for toolset object files.

icvlink – the file format convertor which allows object code to be imported from earlier INMOS toolsets.

2.7.1 Network debugger – `idebug`

The network debugger **idebug** provides post-mortem and interactive debugging for transputer programs. It allows stopped programs to be analysed from their memory image or from image dump files (*post-mortem* debugging) and supports interactive execution of a program using breakpoints (*breakpoint* debugging). Breakpoints can be set on source lines or memory addresses, variables can be inspected and modified, and the program restarted with new values.

idebug provides two debugging environments: a *symbolic* environment which allows a program to be debugged from source code; and the *Monitor page* which allows a program to be debugged at machine level.

2.7.2 Memory dumper – idump

The special debugging tool **idump** is provided to assist with the post-mortem debugging of programs that run on the root transputer. Since **idebug** executes on the root transputer and overwrites the program image, **idump** must be used to save the image to a file which is later read by the debugger.

2.7.3 Librarian – ilibr

The librarian **ilibr** creates libraries of compiled code for use in application programs. Modules generated by **ilibr** are in the same format as code in the standard runtime library and can be used in exactly the same way.

Code written using other compatible toolsets can be mixed with C code in the same library.

2.7.4 Binary lister – ilist

The binary lister **ilist** decodes object code files and displays data and information from them in a readable form. Command line options select the category and format of data displayed.

Examples of the kind of information that can be displayed are symbolic names and attributes, code listing, index data and modular breakdown of libraries, and external reference data.

2.7.5 Transputer simulator – isim

The transputer simulator **isim** provides software emulation of an IMS T425 transputer. Programs configured for single transputers can be run and debugged on the simulator before transferring them to hardware. The debugging environment is similar to that provided by the debugger Monitor page.

Batch mode operation is also supported.

2.7.6 Makefile generator – imakef

The Makefile generator **imakef** creates Makefiles for specific program compilations. Coupled with a suitable MAKE program it can greatly assist with code management and version control.

imakef constructs a dependency graph for a given toolset object file and generates a Makefile in standard format. To allow the tool to work with mixed processor

networks and mixed code programs a standard set of file naming conventions is used during program development.

2.7.7 File format convertor – `icvlink`

The file format convertor **`icvlink`** converts LFF object files generated by earlier INMOS toolsets to standard TCOFF format. TCOFF is a standardised intermediate object file format for transputer programs.

`icvlink` allows existing object code to be used with the INMOS family of TCOFF compilers and toolsets. Files to be converted must be *compiled* files or *linked* object files. The tool is intended to support the importation of code where the source is unavailable and should not be used where code can be recompiled with one of the new compilers.

2.8 EPROM programming

Two tools allow transputer programs to be installed into ROM. These are the EPROM programmer **`ieprom`** and the memory configurer **`iemit`**. An auxiliary tool **`icvemit`** is provided with **`iemit`** for importing memory configuration files generated by the previous INMOS memory configurer tool **`iemi`**.

2.8.1 EPROM programmer – `ieprom`

The EPROM programmer **`ieprom`** converts ROM-bootable files generated by **`icollect`** into a format suitable for input to ROM programmers. Files can be generated for input to ROM loading programs provided for specific EPROMs, or dumped in straight Hex or binary for input to users' own ROM loaders.

2.8.2 Memory configurer – `iemit`

The memory configurer **`iemit`** allows specific memory configurations to be evaluated before running them on hardware. The completed configuration can be included in the **`ieprom`** output file for automatic installation into processor memory. The **`iemit`** support tool **`icvemit`** can be used to convert memory configuration files generated by **`iemi`** (a tool supplied in previous toolsets) to **`iemit`** format.

2.9 Program development using the toolset

The ANSI C toolset is a cross-development system for transputers. Creation of executable code for a transputer or transputer network takes several stages involving the use of specific tools at each stage. Program development is supported by tools which provide facilities for debugging, creating object code libraries, automating the program build, and for importing code from earlier toolsets.

The main stages in developing a program and the tools to use at each stage are listed below.

1 **Write the source.**

Source code can be written using any ASCII editor available on the system. Code can be divided between any number of source files. Source code must conform to the ANSI standard. Source code syntax can be checked prior to compilation by invoking the compiler with the check option.

2 **Compile the source.**

Each source file is compiled using the ANSI C compiler `icc` to produce one or more compiled object files in TCOFF format. Each file must be compiled for the same transputer type or for a transputer class covering several compatible types. Commonly used object code can be combined into libraries using `ilibr`.

3 **Link the compiled units.**

The compiled source files are linked together using `ilink`. This generates a single file called a *linked unit* in which all external references are resolved. The linking operation also links in the library modules required by the program, which are selected by transputer type from the compiled library code. Compiled source files can be generated by any TCOFF compatible compiler.

4 **Configure the program.**

For multitransputer programs a configuration description must be constructed in order to assign linked units to specific nodes on the transputer network and link them by channel variables. The description is processed by the configurer tool `icconf` to produce a configuration data file.

Single transputer programs can also be configured.

5 **Generate a runnable file.**

The configuration data file generated by **icconf** is read by the code collector **icollect** which generates a single executable file for a transputer network. The same tool is used to generate bootable files for single transputer programs directly from linked units.

6 **Load and run the program.**

The executable or *bootable* file is loaded and run on the transputer network down a host link using **iserver**. Once loaded the code begins to execute immediately. The server tool also starts up and maintains the environment that supports the program's communication with the host.

Figure 2.1 illustrates the development process in terms of the architecture of the toolset. The default file extensions assumed and generated by the tools are used to represent source and target files.

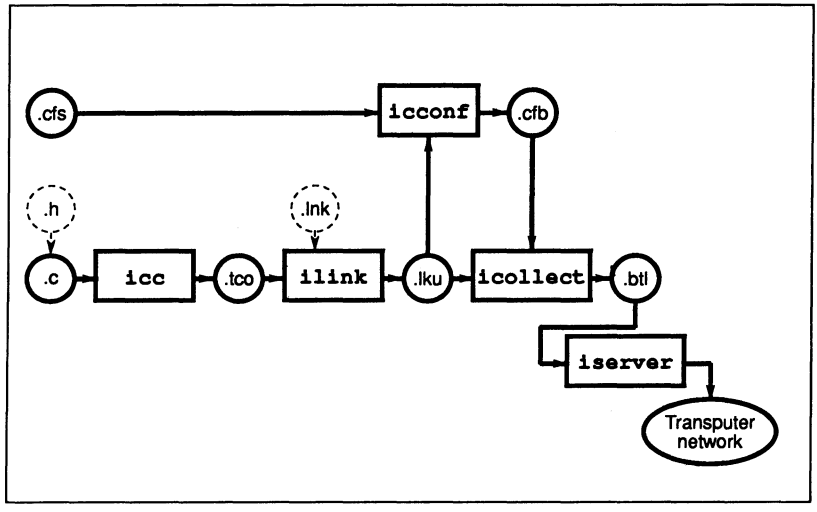


Figure 2.1 Toolset compilation architecture

2.10 Runtime library

The runtime library is a library of compiled C functions that perform common programming operations. The library contains the complete set of ANSI standard functions plus functions to support parallel programming and some non-ANSI extensions.

The concurrency functions are divided into three functional groups: process

management, channel communication, and semaphore handling. The non-ANSI extensions include a set of i/o primitives, a set of short maths functions, functions for retrieving information about the host system, and debugging functions.

A reduced library is available for linking with programs that do not use i/o or i/o dependent functions, for example, code for embedded systems or code that only communicates with other processes on the network and has no direct interaction with the host. The reduced library contains no calls to ISERVER.

2.10.1 Header files

Library functions, like all C functions, must be declared before use. Declarations of library functions with associated constants, macros, and definitions are held in a number of library *header files* to ensure that function declarations are of the correct form and that supporting macros and constants are included. Header files are given the suffix `.h`.

The library header files contain groups of routines collected together according to common usage. For example, routines that control standard i/o operations are grouped in the file `stdio.h`. Most header files also contain definitions of constants and macros that are associated with the functions' use.

Many of the header files and function groupings are defined in the ANSI standard. The library extensions which support concurrency and other non-ANSI operations are also grouped for programming convenience, for example, functions for sending data down channels are grouped separately from those which manipulate semaphores. Similarly, non-ANSI functions such as short maths functions and low level i/o functions are grouped separately. Concurrency functions are in fact grouped into three files covering process handling, channel communication, and semaphore handling.

Some library functions are implemented as macros, and a few are implemented as both functions and macros. The decision about which to use depends on the programming style and personal choice.

2.11 Toolset file extensions

The toolset uses a standard set of file extensions to identify specific source, intermediate, and object files. Certain file extensions are assumed on input, and generated on output if extensions are not specified on the command line. For example, the compiler assumes the suffix `.c` for the input source file and adds the extension `.tco` to the output file unless otherwise specified. The adoption of a standard system allows file extensions to be omitted on the command line and permits host file handling systems to be use manipulate the files. The system

Extension	Description
`.btl`	Bootable code file. Created by the collector tool.
`.btr`	Executable code minus bootstrap information. Used for input to the EPROM tool. Created by the collector tool.
`.c`	C source files. Assumed by **icc**.
`.cfb`	Configuration data file. Created by **icconf**.
`.cfs`	Configuration description. Assumed by **icconf**.
`.lku`	Linked unit. Created by **ilink**.
`.lbb`	Library build file. Assumed by **ilibr**.
`.lib`	Library file. Created by **ilibr**.
`.lnk`	Linker indirect file. Assumed by **ilink**.
`.rsc`	Dynamically loadable code file. Used for calling object modules from source code.
`.tco`	Compiled code file. Created by **icc**.

Table 2.2 Toolset main file extensions

forms an integrated whole and is designed to reflect the architecture of toolset compilation.

The standard set of default file extensions used by the toolset is not mandatory and may be modified according to personal choice (unless **imakef** is to be used to build the program, where a special scheme must be used for mixed processor types and error modes, see below). The standard system has the advantage of ready defaults but may not be readily mapped onto existing development schemes. If you decide to use your own scheme the system should be formal and controlled, particularly where systems are being written by teams of developers.

Some extensions recognised by the toolset are used for convention only and are not interpreted by the tools in any special way. For example, the **.h** suffix for library header files is a C programming convention that has been adopted by the toolset.

The main file extensions are listed in Table 2.2 A full list of all file extensions used by the toolset can be found in appendix A.

The use of default extensions in program development is illustrated in figure 2.1.

File extension scheme required for `imakef`

The Makefile generator **`imakef`** requires a special set of file extensions to be used for compiled and linked object files in order to account for mixed transputer networks and code configured in different error modes. The extensions define the architecture of toolset compilation for **`imakef`** so that it can trace file dependencies and construct the proper commands for making target files.

For details of the file extensions that you must use with the **`imakef`** tool see section 22.4.

2.12 Error reporting

All errors are reported in a standard format containing the name of the tool, a severity level, and some explanatory text explaining why the error occurred. Errors found in files or the file system may also generate a filename and line number. Standardisation of the format is designed to improve error reporting and to support automated error handling by host system utilities.

For example:

Warning–icc–prog.c (25) inventing 'extern int foo();'

Note: Messages that are part of the normal operation of the tool, for example, syntax errors generated by the compiler and messages from the debugger and simulator tools, are not required to conform to the standard and may be displayed in special formats. The formats are designed to be appropriate for the tools' purpose and will become familiar with use.

Details of the standard format can be found in appendix A.

2.13 Host dependencies

The ANSI C toolset can be hosted on several platforms, and is designed to blend in as far as possible with each host operating system. Source and object code is portable between all systems.

The toolset is available for the following systems:

- IBM PC and NEC PC running MS-DOS
- VAX running VMS
- Sun 3 running SunOS

- Sun 4 running SunOS

Differences between the various platforms are minor and reflect the 'flavour' of the particular operating system. This leads to minor differences between them in the areas of command line syntax and characters allowed in filenames. Some installation issues are also host dependent, for example the setting of environment variables and the definition of search paths. These are covered in detail in the Delivery Manual that describes product installation and system setup, and are only described briefly here.

Host system dependencies are as far as possible made invisible to the user. The few differences are some minor variations in command line syntax, host-specific library routines, directory names, and environment settings such as search paths and global variables. Each is described briefly below.

Command line syntax

The major difference between the various host implementations is the use of the host system option prefix. For UNIX based toolsets the prefix character is the dash '-'; for IBM PC and VAX/VMS based toolsets the prefix is the forward slash '/'. For consistency between implementations, the case of options is also not significant. Other command line syntax conventions are identical in all four implementations and are described in appendix A.

2.13.1 Host-specific library

All library functions supplied with the toolset are host independent except for the functions declared in `dos.h` which are specific to DOS. The DOS functions are supplied with all host versions of the toolset.

Care should be taken in the use of these functions in application programs. Programs which use them will not be portable across all four systems.

2.13.2 Filenames

Filenames, with or without the full directory path, conform to the normal host system conventions except that characters which can be interpreted as directory separators must not be used in the filename part. Prohibited characters are: colon `:`, forward slash `/`, backslash `\`, and closing square bracket `]`.

2.13.3 Search paths

All tools which use or generate filenames use a standard mechanism for locating files on the host system. The same mechanism is used in all operating system versions of the toolset. Briefly, the search mechanism is based on a list of directories to be searched in sequence. If a directory path is specified only this directory is searched. Relative pathnames are treated as relative to the current directory. If no directory path is specified the current directory is searched followed by the directories specified in the **ISEARCH** environment variable. Details of how to set up environment variables on your system can be found in the Delivery Manual that accompanies the release.

Details of the mechanism can be found in appendix A.

2.13.4 Environment variables

The toolset uses a number of environment variables on the host system. Use of these variables is optional but if defined they will affect the behaviour of the tools on your system.

Variable	Meaning
ISEARCH	The search path i.e. the list of directories that will be searched if a full pathname is not specified. Pathnames must be terminated by the standard directory separator character for the system. Used by all tools that read and write files.
ITERM	The file that defines terminal keyboard and screen control codes. Used by **idebug** to define symbolic function keys.
IBOARDSIZE	The size (in bytes) of memory on the transputer board. Used by **iserver**.
TRANSPUTER	The address at which the transputer board is connected to the host. Used by **iserver**.
IDEBUGSIZE	The size (in bytes) of memory connected to the root transputer. Used by **idebug**.
*toolname***ARG**	Default command line arguments. (Applies to certain tools only. See section 2.13.5.)

The exact commands used to define environment variables depend on the operating system. For example, on the IBM PC they are defined using the **set** command; on VAX systems running VMS they can be set up either as logical names or as VMS symbols. Examples of how to set up environment variables can be found in the Delivery Manual that accompanies the release.

For **IBOARDSIZE** and **IDEBUGSIZE** the value can be given in decimal or hexadecimal format. Hexadecimal numbers must be preceded by '**#**' or '**$**'.

Leading and trailing spaces are prohibited in both variables.

Note: If **IBOARDSIZE** is specified incorrectly, for example as a character or string, the system defaults to a board size of 0 (zero) and the program cannot be run. If **IBOARDSIZE** is explicitly set to a very small value a similar error may occur.

2.13.5 Default command line arguments

An environment variable can be defined on the system to specify a default set of command line arguments for certain tools. The variable name must be defined in upper case and is constructed from the tool name by appending the letters '**ARG**'. For example, the variable for **icc** is **ICCARG**.

Tools for which a default command line can be defined, and the variables used to define them, are listed below.

Tool	Variable
icc	**ICCARG**
ilink	**ILINKARG**
icconf	**ICCONFARG**

Tool	Variable
ilibr	**ILIBRARG**
ilist	**ILISTARG**
icvlink	**ICVLINKARG**

Command line parameters must be specified within each variable using the specific syntax required by each tool.

3 Getting started

This chapter outlines how to compile, link, and run simple C programs on a transputer, using sample programs provided in subdirectories of the toolset **examples** directory.

3.1 Outline procedure

In order to create a program that will run on a transputer or transputer network you must:

1 Compile each source file using the ANSI C compiler **icc**. By default the compiler codes for a T414 transputer.

2 Link the separately compiled object files with each other, and with the libraries that they use, using the linker **ilink**.

3 Configure the program for the transputer or transputer network. For multitransputer programs a configuration description must be written for processing by the configurer and the resulting configuration data file passed to the collector in order to generate a loadable or **bootable** file. For single transputer programs the collector is used to bootstrap the linked unit directly by invoking it with a special option.

4 Load the bootable program onto the network using the host file server tool **iserver**. Bootable programs are self-starting and begin to run immediately they are loaded into transputer memory.

3.2 Running the examples

In the following examples programs are compiled for the default processor type T414. For other transputer types, for example, processors in the IMS T8 group, the program must be compiled for the specific transputer target or a transputer class by giving the appropriate option on the compiler command line. Details of transputer types and the cross compatibilities of processor types and classes can be found in section 5.3.

3.2.1 Sources

The sources of all the examples are held on the toolset **examples** subdirectory. To use the example command lines either move to this directory or place a copy of the file in your current directory.

3.2.2 Example command lines

Where necessary, example command lines are duplicated for different host versions of the toolset; the '-' switch character is used in command lines for UNIX-based toolsets and the '/' character is used in commands for MS-DOS and VMS based toolsets. When reproducing the examples you should use the appropriate command line for your host system.

3.2.3 Using the simulator

If no transputer hardware is available the examples can be run using the T425 simulator **isim**. If the simulator is used the appropriate **isim** command line should be substituted for the **iserver** command line in all example procedures. Details of how to invoke the simulator tool can be found in chapter 24.

3.3 A simple sequential program

The following procedure shows how to build and run a simple 'Hello World' program. The program is held in the source file **hello.c** on the **simple** examples subdirectory.

3.3.1 Compiling

To compile the program type:

```
icc hello
```

The compiler assumes a **.c** extension. If the source file contains no errors, the compiled object file **hello.tco** is produced.

3.3.2 Linking

The compiled object file must be linked with the runtime library and startup code using the linker tool **ilink**. Use one of the following commands:

```
ilink hello.tco -f startup.lnk          (UNIX)
ilink hello.tco /f startup.lnk          (MS-DOS and VMS)
```

This produces the linked unit **hello.lku**. As no output file is specified the file is named after the input file and the default link extension **.lku** is added.

The '**f**' option specifies the standard C startup file containing commands and directives to **ilink**. The file is in standard linker indirect file format and contains

all the startup information required for a C program using the full runtime library. This includes access points for the library, and no libraries need be specified on the command line.

3.3.3 Configuring

The linked unit must now be configured for the transputer. Because the program is to be run on a single processor the configurer is not required and **icollect** can be used directly. To bootstrap a single linked unit the '**t**' option must be specified:

```
icollect hello.lku -t                    (UNIX)
icollect hello.lku /t                    (MS-DOS and VMS)
```

This creates the file **hello.btl** which can be loaded and run on a single transputer.

3.3.4 Loading

iserver is used to load the bootable file down the host link into transputer memory where it begins to execute immediately:

```
iserver -sb hello.btl                    (UNIX)
iserver /sb hello.btl                    (MS-DOS and VMS)
```

The program runs and displays the greeting "**Hello World**".

3.4 A parallel version

The example program **parhello.c** on the **simple** examples subdirectory is a parallel version of the "Hello world" program, using separate parallel processes to output each word of the greeting. A time delay is built into one of the processes to demonstrate their independence.

The example produces the same output as the sequential program and is included here in order to introduce a simple working example of a parallel C program. Parallel programming is described in greater detail in chapter 4.

To run the example parallel program compile, link, configure, and load the pro-

gram in the normal way:

```
icc parhello

ilink parhello.tco -f startup.lnk          (UNIX)
ilink parhello.tco /f startup.lnk          (MS-DOS and VMS)

icollect parhello.lku -t                   (UNIX)
icollect parhello.lku /t                   (MS-DOS and VMS)

iserver -sb parhello.btl                   (UNIX)
iserver /sb parhello.btl                   (MS-DOS and VMS)
```

The program prints the word "Hello" followed after a short delay by "world".

The overall construction of the program can be deduced from the program listing. Briefly, processes to be run in parallel are defined as separate C functions, space is allocated for the process structures, and the functions are started up in parallel. A comparison of the source code for sequential and parallel versions can be instructive.

3.5 Separate compilation

Larger programs are often built from a number of separately compiled source files. The following example shows how to build and run the parallel "Hello World" program from three source files. The program sources are held on the **simple** examples subdirectory.

The main program in **main.c** calls two independently compiled parallel processes **hellof** and **worldf** which each print one word of the greeting.

To run the program first compile each source file:

```
icc main

icc hellof

icc worldf
```

This creates three compiled object (**.tco**) files. These are then linked with each

other to produce the single linked unit **main.lku**:

```
ilink main.tco hellof.tco worldf.tco -f startup.lnk
```
(UNIX)
```
ilink main.tco hellof.tco worldf.tco /f startup.lnk
```
(MS-DOS and VMS)

The linked unit is then bootstrapped in the normal way for a single transputer using **icollect**, and loaded into transputer memory using **iserver**:

```
icollect main.lku -t
```
(UNIX)
```
icollect main.lku /t
```
(MS-DOS and VMS)

```
iserver -sb main.btl
```
(UNIX)
```
iserver /sb main.btl
```
(MS-DOS and VMS)

3.6 A simple configuration example

Linked units can be configured on processor networks by processing a *configuration description* with **icconf**. The file **hello.cfs** on the **config** examples subdirectory contains a configuration description for a "Hello World" program along with the source file. (The source file is not the same as the one used in the non-configured examples).

The description configures the program for a single processor, which is treated by **icconf** in the same way as any multiprocessor network.

Either move to the **config** examples subdirectory or copy the "Hello World" source file **hello.c** and the "Hello World" configuration description **hello.cfs** from the directory into a working directory.

Compile and link the source file as in the previous examples to produce a linked unit. Now run **icconf** on the **hello.cfs** file. This produces the configuration binary file **hello.cfb**. Run the collector on this file, this time omitting the '**t**' option (only required when the input file is a linked unit). Then load the program in the normal way using **iserver**.

The sequence of commands is illustrated below.

```
icc hello

ilink hello.tco -f startup.lnk          (UNIX)
ilink hello.tco /f startup.lnk          (MS-DOS and VMS)

icconf hello.cfs

icollect hello.cfb

iserver -sb hello.btl                   (UNIX)
iserver /sb hello.btl                   (MS-DOS and VMS)
```

4 Parallel processing

4.1 Introduction

Parallel processing is widely accepted as an important way of improving software performance on any given processor architecture. The transputer supports parallel processing directly by incorporating into its design a process scheduler which is responsible for scheduling parallel tasks, and by providing the means for connecting processors (transputer links) to create a processor network.

ANSI C supports concurrent programming by runtime library extensions which allow C functions to run in parallel and communicate via channels. The extensions provided consists of new type definitions for processes and channels and a set of library functions for process, channel, and semaphore handling.

Semaphore-based communication is also supported.

4.2 Abstract model

Parallel processing in transputer based systems is based on the idea of *Communicating Sequential Processes* (CSP) developed by by Professor C.A.R. Hoare.

CSP is an abstract generalised model of concurrency based on the idea of independently executing processes exchanging data with each other via one way connections called channels. The model can be used to describe software applications in an intuitive way reflecting the parallelism of the real world.

Concurrent processing in ANSI C conforms to the CSP model. Concurrent C processes are independent, can be nested within each other, and are linked together by channels. Any C function can be defined as a concurrent process using a special set of functions provided in the runtime library.

Figure 4.1 illustrates the main elements of the CSP model. Processes can be nested with one another, and can communicate either unidirectionally (one process passing data to another) or bidirectionally (two processes exchanging data and working in a cooperative manner). In real applications processes normally communicate with at least one other process in the system.

4.2.1 Processes

Processes are the main elements of the CSP model. A process describes the behaviour of a discrete, separable component of an application; it may consist of other processes, sequential operations, or any combination of these. Appli-

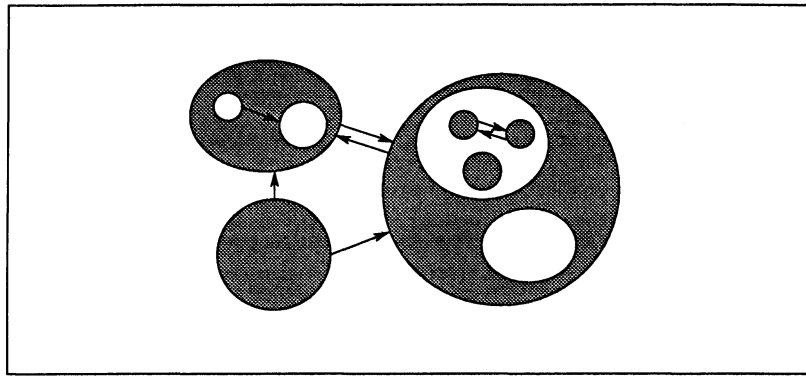


Figure 4.1 Communicating sequential processes

cations can be broken down into any number of processes, and processes can be mapped onto any network of transputers.

4.2.2 Channels

Channels are the connections between processes through which information and data are exchanged. Channels are point-to-point unidirectional connections, that is, they connect only two processes, and the transfer of data is one way. Processes which exchange messages and data with each other must do so via a pair of channels. Channels in real systems are often paired in this way to enable processes to cooperate in a task.

An item of data is always acknowledged by the receiving process before the next item is passed.

In CSP, channels have two functions. They provide the communication path between independently executing processes, and serve to synchronise the communication between the two processes. Items of data must always be acknowledged by the receiving process and the sending process always waits for the acknowledgement. In the same way processes which send data cannot do so until the receiving process is ready. In this way synchronisation between the two processes is assured; no data is passed until both partners in the operation are ready.

4.3 Semaphores

Support for semaphores, though not a part of the CSP model, is provided in the toolset for those who wish to develop parallel programs in the traditional manner using semaphores. Semaphores are efficiently implemented within the toolset using channel functions, and are therefore subject to a slightly greater overhead than if the intrinsic synchronising ability of channels were used directly.

4.4 Parallel processing and transputers

The transputer has been designed to support parallel processing and the construction of multiprocessor environments. The device architecture and instruction set reflect the CSP model and make it easy to implement in high level languages. ANSI C takes full advantage of this ability, providing a parallel programming environment optimised for the transputer, but retaining all the features of the C language.

Each transputer separately supports parallel processing. Processes can exchange data and synchronise their activity by a scheduling system built into the hardware of the processor and requiring no complex programming. The system automatically time shares the CPU between processes and requires no extra input from the programmer. Communication between processes is achieved via channels implemented as words in on-chip memory.

4.4.1 Multitransputer networks

Processes can also run on separate transputers and communicate with each other using channels implemented through processor links. Each transputer contains four INMOS communication links through which processors exchange data and information. This ability to be cross-connected enables the transputer to be used as the basic component in the construction of processor networks. Specific arrangements of transputers can be designed for particular software tasks, and large networks of transputers can be used to build distributed processing supercomputers.

4.4.2 Multitransputer programming

Software processes can communicate as readily down transputer links as they can across channels on the same processor. This allows applications to be written without being constrained by a fixed topology; a program can be developed on a single transputer and ported to the target network when the program is fully developed and bug-free. The final code can be reinstalled on a new target simply by writing a new configuration description, generating an executable file, and

loading it onto the network. No modification of the original code or recompilation of source is required.

Figure 4.2 shows how three software processes, separately compiled and linked, could be configured to run on a single processor or on several processors linked together in a simple linear network.

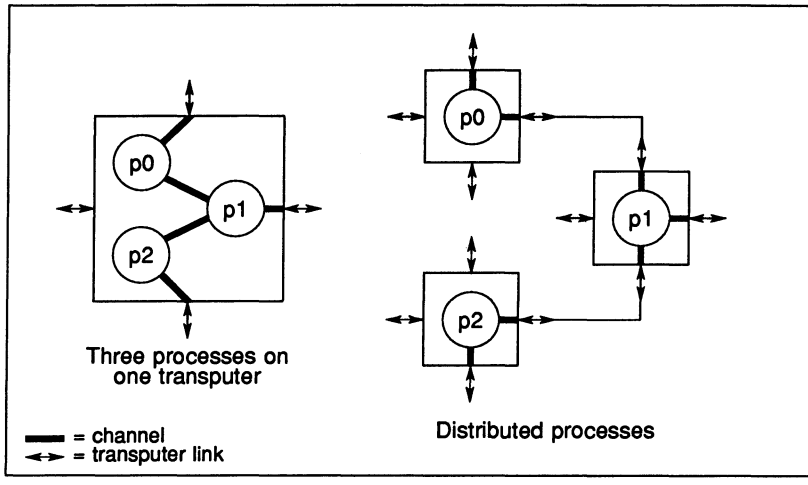


Figure 4.2 Distributing processes

4.4.3 Instruction set

Transputers have been designed to support the ideas of parallel processing and make them easy to implement in high level languages. There is direct support in the transputer instruction set for process control and management.

Process control

The transputer provides direct instructions for setting up, starting, pausing, and terminating parallel processes. Processes run at one of two priorities – high or low; high priority processes have priority access to the processor and will always be executed in preference to any low priority process running concurrently on the same processor.

Process selection

The transputer instruction set includes direct support for selection of the first ready process from a series of of inputs, making polling of data channels easy.

Process timing

The transputer contains high and low priority clocks, which can be used to implement delayed execution of processes. Specific instructions are provided to delay execution of a process for a specified time period, or until a specified time.

4.5 ANSI C

ANSI C takes full advantage of the advanced concurrency features of the transputer and like the high level transputer language occam implements the CSP model. Concurrency is supported by library extensions consisting of three new data types and a set of library functions and macros. Together these implement the parallel model. A set of routines for synchronising processes using semaphores is also provided.

4.5.1 Library support

The runtime library functions are accessed in the same way as all other C library functions by including the appropriate header file in the program. Process, channel, and semaphore support functions are declared in three separate header files.

The concurrency functions are designed as a base set of functions which can either be used in their basic form or as building blocks for higher level routines. For example, a high level package might wish to implement features such as process multiplexing and complex channel protocols using functions from the basic set.

4.5.2 New data types

Three new data types complete the concurrency support. Data structures aare used to hold data about processes and semaphores, and a pointer type is used to implement channels.

- `Process`. A structure type that holds information about each declared process.
- `Channel`. A pointer type used to implement channels. In accordance with the CSP model, channel variables represent unidirectional communication links between two processes. `Channel` is a pointer to type `void`.
- `Semaphore`. A structure type that holds information about a semaphore.

Parallel processes are created by linking a function definition to a predeclared process structure, and are then initialised, started, and run using routines from the concurrency library.

Channels between processes are created simply by declaring a variable of type `Channel*` at an appropriate point in the program. Channel input and output functions are then used to pass data. It is the responsibility of the programmer to ensure that data sent by one process is received by another; separate functions exist for input and output and the two must be paired for communication between two processes to take place.

Semaphores are declared using either the semaphore initialisation function or a macro that performs a similar action. Semaphores are then acquired and released by calls to two separate functions. Semaphores can be used to synchronise the activity of low with high priority processes.

4.6 Concurrency functions

The concurrency functions implement the following parallel processing operations:

- Process setup, startup, and scheduling
- Ready input selection
- Channel communication
- Semaphores.

The main parallel processing functions are declared in the header files `process.h` and `channel.h`. Declarations of functions for semaphore handling can be found in `semaphor.h`.

The following sections describe the process, channel, and semaphore functions.

4.7 Processes

Processes are defined in the same way as regular C functions, but with a fixed first parameter. The first parameter to a new function which will be started as a process *must* be a pointer to its own `Process` structure. Parameters to the function follow the fixed process pointer in the normal way.

Processes are instantiated by a call to `ProcAlloc`, using the function name as the link to the process structure. Once allocated, the process is started

using **ProcRun** or one of its variants, **ProcPar** or one of its variants, or **ProcPriPar**.

An example of the creation and instantiation of a process is shown below.

```
void newproc(Process *p, int arg1, int arg2, int arg3)
{
  p = p;

  /*  ...process code */
}

int main()
{
  /*  Declare pointer to process structure */

  Process *x;

  /*  Declare parameters */

  int pa1, pa2, pa3;

  /*  Allocate process; check for non-allocation */

  if  ((x = ProcAlloc(newproc,0,3,pa1,pa2,pa3)) == NULL)
    abort();

  /*  Start process running */

  ProcRun(x);

  /*  Rest of code executes
      in parallel with 'newproc' */

}
```

4.7.1 Unused process pointer

The compiler generates a warning message indicating an unused process pointer each time a process pointer is passed to a function. To prevent the message being generated the process pointer should be assigned to itself within the function using a statement of the form `p = p;`. Process code which does not assign the pointer in this way will still compile and run normally but the 'unused pointer' message will be generated as the process is compiled.

Warning: The process pointer passed through to a function is used internally by the concurrency software and must never be changed. If it is modified in any

way the results are undefined.

4.7.2 Process initialisation

Two functions allocate and initialise parallel processes. A third function is provided to allow parameters to be altered in an existing process. The three functions and their parameters are listed below.

Function		Parameters
`Process`	`ProcAlloc`	`(void (*func)(), int size, int nparam, ...)`
`int`	`ProcInit`	`(Process *p, void (*func)(), int *ws, int wssize, int nparam, ...)`
`void`	`ProcParam`	`(Process *p, ...)`

`ProcAlloc` reserves memory space for a process and initialises the `Process` structure using the lower level routine `ProcInit`. `ProcInit` can also be used directly to initialise a process for which the memory space has already been reserved by the programmer.

The ancillary function `ProcParam` allows parameters to be changed in an existing (previously allocated) process. It must be called before the process is started up.

`ProcAlloc` takes a pointer to the function code, allocates a stack frame for the process, and sets up the function's parameters. A pointer to the process structure is returned.

The stack size is specified in the `size` parameter. If `size` is specified as zero a default stack size of 4K for 32-bit machines and 1K for 16-bit machines is used instead. If insufficient stack space is allocated for the required number of parameters, the stack is extended. `ProcAlloc` returns a pointer to the process structure (`Process*`).

Processes set up using `ProcAlloc` share the same global data space and therefore access the same static and external variables. Private data space for a process must be allocated using `auto` variables. In addition `ProcAlloc` uses the standard functions `malloc` and `free` to allocate and deallocate space from the heap and as a result all C processes share the same heap space. If an attempt is made to allocate stack space from an array of type `auto`, an error is reported.

All calls to **ProcAlloc** should be followed by a check for successful allocation, and the NULL result (allocation unsuccessful) should be handled in an appropriate way.

ProcInit takes a pointer to an existing **Process** structure and a pointer to the stack space to be used. It then initializes the process structure and workspace for the function according to its workspace requirement and process parameters. **ProcInit** is the lower level routine used by **ProcAlloc**. **ProcInit** returns a value indicating success or failure. The number of parameters indicated by **nparam** excludes the compulsory process pointer.

Note: Processes must always be allocated before use. If this is not done the same memory space may be referenced on behalf of the same process. In this context allocation can be performed by **ProcAlloc** or **ProcInit**.

ProcParam can be used to modify the parameters of an already allocated process. It returns no result.

Note: Care should be taken when setting up processes and changing parameters in concurrently executing processes. If **ProcAlloc**, **ProcInit**, or **ProcParam** are used from two parallel processes to initialise the same process the results may be unpredictable because there may be contention for the process structure.

4.7.3 Freeing stack and workspace

Two functions **ProcAllocClean** and **ProcInitClean** are provided to free stack and workspace after a process has completed. The functions and their parameters are listed below.

Function	Parameters
`void ProcAllocClean`	`(Process *p)`
`void ProcInitClean`	`(Process *p)`

ProcAllocClean is used for processes initialised using **ProcAlloc**, and **ProcInitClean** for processes initialised using **ProcInit**.

Note: For both functions the process must have been started synchronously (by **ProcPar** or **ProcParList**), and must have already terminated (**ProcPar** or **ProcParList** must have returned).

4.7.4 Process execution

A set of functions is provided for executing processes asynchronously (**ProcRun** and related functions) or synchronously (**ProcPar** and others). Functions are provided in the first group to start processes at high or low priority, and in the second group to start many processes in a single call or to start a pair of processes at high and low priority.

Note: A process may only be started once. If the same process pointer is passed as an argument to more than one function running as a process unpredictable effects can occur.

Function	Parameters
`void ProcRun`	`(Process *p)`
`void ProcRunHigh`	`(Process *p)`
`void ProcRunLow`	`(Process *p)`
`void ProcPar`	`(Process *p1, Process *p2, ...)`
`void ProcParList`	`(Process **plist)`
`void ProcPriPar`	`(Process *phigh, Process *plow)`

Unsynchronised processes

ProcRun, **ProcRunHigh** and **ProcRunLow** start processes which execute independently of the process from which they are called. The initiating process cannot determine or alter the state of the process except through an explicit communication path (usually a channel) that the programmer establishes.

ProcRun starts a process at the same priority as the calling process. **ProcRunHigh** and **ProcRunLow** start processes at high and low priority respectively.

Unsynchronised processes (those started with the **ProcRun**, **ProcRunHigh**, and **ProcRunLow** functions) run independently of the main program and may continue when the main program terminates. To ensure that processes do not access the server when it has already been terminated, processes can be coupled to the main program using channels. The main program will then wait until all other processes are finished before it terminates the server. Alternatively, **ProcPar** can be used to force synchrony on a group of processes.

Synchronization channels are channels on which an output to the main program

is performed as the last action in a process. This forces the main program to wait until all other processes are completed and ensures clean termination of a program. An example of how to use synchronisation channels is shown below.

```
#include <process.h>
#include <channel.h>

void p1(Process *p, Channel *synch)
  {
   p = p;
   /* ...process code */

   ChanOutInt(synch, 1);
   /*
      Sends integer to main program to signal
      completion
   */
  }

int main()
  {
   Process *p;
   Channel *synch; /* Synchronization channel */

   if ((synch = ChanAlloc()) == NULL
     {
      /* Call channel error handler */
     }

   if ((p = ProcAlloc(p1, 0, 1, synch)) == NULL)
     {
      /* Call process error handler */
     }

   ProcRun(p);

   ChanInInt(synch);
   /*
      Receives completion signal from process p1
   */

  } /* Program can terminate safely */
```

Synchronised processes

`ProcPar`, `ProcParList`, and `ProcPriPar` each start a group of processes.

Control is returned to the process from which the function was called when all processes in the group have terminated.

ProcPar executes a group of processes at the current priority. The list of processes must be terminated by NULL. **ProcParList** takes an array of pointers to processes and executes them at the current priority. The list must be terminated by NULL. **ProcPriPar** takes two processes and executes them at high and low priority. The first process in the list is executed at high priority.

Note: **ProcPriPar** can only be called from a low priority process; if the function is called from a high priority process a runtime error occurs and the program is aborted.

4.7.5 Process timing and scheduling

Routines are provided for delayed execution, the timed suspension and rescheduling of processes, and the termination of processes before normal completion.

Function	Parameters
`void ProcAfter`	`(int time)`
`void ProcWait`	`(int time)`
`void ProcReschedule`	`(void)`
`int ProcGetPriority`	`(void)`
`void ProcStop`	`(void)`
`int ProcTime`	`(void)`
`int ProcTimePlus`	`(const int time1, const int time2)`
`int ProcTimeMinus`	`(const int time1, const int time2)`
`int ProcTimeAfter`	`(const int time1, const int time2)`

Process timing

Execution of a process can be delayed until a specified time using the **ProcAfter** function, and suspended for a specified time using **ProcWait**.

Process scheduling

ProcReschedule reschedules a process, that is, places it at the end of the process queue. This can be used to implement a 'busy wait' on a resource.

ProcGetPriority returns the execution priority of a process (1 for a low priority process and 0 for a high priority process. Macros **PROC_LOW** (1) and **PROC_HIGH** (0) are defined within the process library.).

ProcStop permanently deschedules a process. It is used to stop a process before normal completion. Stopped processes cannot be restarted.

4.7.6 Clock time

ProcTime returns the current value of the clock. The selection of the high or low priority clock depends on the priority of the process from which the function is called.

ProcTimePlus returns the result of adding **time1** to **time2**. **ProcTimeMinus** returns the result of subtracting **time2** from **time1**. Both functions use modulo arithmetic – there is no overflow checking and the values are cyclic.

ProcTimeAfter determines the relationship of one transputer clock value to another. It returns 1 if **time1** is after **time2**, otherwise 0 (zero).

4.7.7 Input alternation

Six routines are provided to allow for the selection of a ready channel from multiple parallel inputs. Separate versions of the routines are provided to deal with lists of channels.

Function	Parameters
`int ProcAlt`	`(Channel *c1, Channel *c2, ....)`
`int ProcAltList`	`(Channel **clist)`
`int ProcSkipAlt`	`(Channel *c1, Channel *c2, ....)`
`int ProcSkipAltList`	`(Channel **clist)`
`int ProcTimerAlt`	`(int time, Channel *c1, Channel *c2, ....)`
`int ProcTimerAltList`	`(int time, Channel **clist)`

ProcAlt, **ProcSkipAlt**, and **ProcTimerAlt** take as input a list of pointers to channels terminated by **NULL**. Similarly, **ProcAltList**, **ProcSkipAltList**, and **ProcTimerAltList** take as input an array of channel pointers terminated by **NULL**.

4.7.8 Simple alternation

ProcAlt and **ProcAltList** suspend the current process until one of the channel arguments is ready to input. On completion, the functions return an index into the parameter list indicating the ready channel.

For example, the following code sets **i** to 0, 1, or 2, according to which of the three channels becomes ready first:

```
int i;
Channel *c0, *c1, *c2;

i = ProcAlt(c0, c1, c2, NULL);
```

Both **ProcAlt** and **ProcAltList** require at least one input parameter; if the parameter list is empty an error is generated.

4.7.9 Polling several inputs

ProcSkipAlt and **ProcSkipAltList** check a series of channels without blocking the current process. If one of the channels is ready to input an index into the parameter list is returned. Both functions return immediately with a special code value and do not wait for a channel to become ready.

4.7.10 Timed input

ProcTimerAlt and **ProcTimerAltList** block the current process until one of the channels is ready for input or until a specified time is reached. If a channel becomes ready before the timeout occurs an index into the parameter list is returned, otherwise the timeout code is returned.

4.7.11 Example of use

All six alternation routines return an index to the ready channel rather than the data itself and must be followed by a statement which performs the input.

```
int i;
char buffer[LENGTH];
Channel *c0, *c1, *c2;

i = ProcAlt(c0, c1, c2, NULL);
switch(i)
  {
   case 0: /* channel c0 ready */
   ChanIn(c0, buffer, LENGTH); breakj;

   case 1: /* channel c1 ready */
   ChanIn(c1, buffer, LENGTH); breakj;

   case 2: /* channel c2 ready */
   ChanIn(c2, buffer, LENGTH); breakj;

   default:
   error_handler(); breakj;
  }
```

4.8 Channel communication

Routines are provided for the passing of bytes and integers on channels (**ChanIn**, **ChanOut**, and others), for implementing safe channel protocols (extraordinary link handling), and for allocating and resetting channels.

Communication between processes is effected by passing data on variables of type **Channel**. Functions are provided to allocate, initialise, and reset channels, to support input and output of characters, integers, and untyped data, and to assist with establishing reliable protocols (extraordinary link handling).

Channel input and output functions must be paired for two processes to communicate and exchange data.

4.8.1 Transputer link addresses

Each link on a transputer is associated with an input and an output channel address. Transputer link addresses are defined in the library header file `channel.h`.

4.8.2 Channel allocation, initialisation, and reset

Function		Parameters
`Channel`	`*ChanAlloc`	`(void)`
`int`	`ChanReset`	`(Channel *c)`
`void`	`ChanInit`	`(Channel *c)`

`ChanAlloc` reserves space from the heap for the channel, initialises the channel and returns a pointer to it. If space cannot be allocated `ChanAlloc` returns NULL.

Note: All calls to `ChanAlloc` should be followed by a check for successful allocation, and any NULL result (allocation unsuccessful) handled appropriately.

`ChanReset` resets a channel to its quiescent (non-communicating) state, returning either a descriptor to the process waiting to communicate, or the value `NotProcess_p` which indicates the previous communication completed successfully and the channel is free. `NotProcess_p` is a macro defined in the header file `channel.h`.

`ChanInit` initialises a channel by writing the the value `NotProcess_p` into the channel word.

Note: Channels between processes running on the same transputer (*soft* channels) must always be allocated before use. If this is not done the same memory space may be referenced on behalf of the same channel. In this context allocation can be peformed by `ChanAlloc` or `ChanInit`.

4.8.3 Channel input and output

`ChanOut` and `ChanIn` perform the basic operation of passing bytes on a channel. The data can be of any type or size but the number of bytes must be specified. Typed data should be broken down into individual bytes for transmission and retyped on input.

`ChanOutChar` and `ChanInChar` are similar except that they pass single characters and no byte count is required. `ChanOutInt` and `ChanInInt` are similar except that they pass single integers.

Function		Parameters
`void`	`ChanOut`	`(Channel *c, void *cp, int cnt)`
`void`	`ChanIn`	`(Channel *c, void *cp, int cnt)`
`void`	`ChanOutChar`	`(Channel *c, char ch)`
`char`	`ChanInChar`	`(Channel *c)`
`void`	`ChanOutInt`	`(Channel *c, int n)`
`int`	`ChanInInt`	`(Channel *c)`

`ChanOut` and `ChanIn` are used to transfer data of any type. Each take as parameters the channel to be used, a pointer to the data to be passed, and the number of bytes of data.

4.8.4 Reliable channel protocols

The standard input and output channel functions do not attempt to recover from physical link failure. If there is a faulty connection between two processors, processes waiting for communication on that link can never complete successfully.

Four functions are provided to allow recovery from link failure. The functions can be used as they are, or in the definition of higher level functions for reliable channel protocols.

Function	Parameters
`int ChanOutTimeFail`	`(Channel *chan, void *cp, int cnt, int time)`
`int ChanOutChanFail`	`(Channel *chan, void *cp, int cnt, Channel *failchan)`
`int ChanInTimeFaill`	`(Channel *chan, void *cp, int cnt, int time)`
`int ChanInChanFail`	`(Channel *chan, void *cp, int cnt, Channel *failchan)`

The functions are essentially the same as **ChanOut** and **ChanIn** except that the communicating process becomes rescheduled after a specified timeout (...TimeFail functions) period or after receiving a communication on a special reset channel (...ChanFail functions). The ...ChanFail functions allow a communication to be aborted by a separate process set up to monitor the integrity of the link.

The reliable channel functions offer somewhat more overhead than their **ChanIn** and **ChanOut** counterparts. They are designed for establishing the integrity of a link between two unfamiliar processors rather than as the standard method of communicating data.

The reliable communication routines do not attempt to reestablish communication between two processes. This is a problem which is properly addressed at the application level.

4.8.5 Semaphores

Semaphore handling routines are provided for programmers who wish to write traditional parallel code based on the acquisition and release of tokens. The routines are used within the implementation of INMOS C to parallelize certain functions in the standard i/o library.

Semaphore support is included for implementing special operations rather than for implementing normal parallelism, for which the concurrency functions are provided. The standard process and channel functions provide the best way of using the transputer hardware to execute parallel code.

Semaphores can be used to synchronise high with low priority processes.

Note: Semaphores are mapped by the compiler onto standard channel processing functions. This involves some overhead and for maximum efficiency the functions can be used directly.

Use of semaphores by the library

Semaphores are used in the language implementation to parallelize some library routines. For instance, they are used in the implementation of **malloc**, **free**, and **realloc** to prevent the heap being corrupted by simultaneous calls from concurrently executing processes.

File descriptors used internally by the compiler also use semaphores.

4.8.6 Semaphore allocation

Two functions and a macro are provided to set up and initialise semaphores. All perform the same basic operation of creating a semaphore and their use depends on personal choice.

Function		Parameters
`Semaphore`	`*SemAlloc`	`(int initvalue)`
`void`	`SemInit`	`(Semaphore *sem, int initvalue)`
`SEMAPHOREINIT`		`(int initvalue)`

In all three cases **x** is the value of the semaphore on creation.

If **SemAlloc** fails to allocate space for the semaphore a NULL result is returned. All calls to **SemAlloc** should be followed by a check for successful allocation, and the NULL result (allocation unsuccessful) should be handled appropriately.

SEMAPHOREINIT can be used to initialise the semaphore at declaration time. It is particularly useful for static semaphores.

Examples

```
1 #include <semaphor.h>
  Semaphore *newsem;
  newsem = SemAlloc(x);

2 #include <semaphor.h>
  Semaphore sem;
```

```
SemInit (&sem, x);

3 #include <semaphor.h>
  Semaphore sem = SEMAPHOREINIT(x);
```

4.8.7 Semaphore handling

Two routines synchronise the acquisition and release of semaphores:

Function	Parameters
`void SemWait`	`(Semaphore *sem)`
`void SemSignal`	`(Semaphore *sem)`

`SemWait` allows the calling process to acquire the semaphore. If the semaphore is already in use (that is, has the value 0), the current process is suspended and placed on a semaphore queue; if the semaphore is free the semaphore is *acquired*, that is, incremented, and the process continues to execute.

`SemSignal` releases the semaphore and runs the first process waiting on the semaphore queue. If there is no process waiting the semaphore is incremented.

4.9 Parallel programming examples

The following three examples are parallelized versions of the "Hello World" program. They are designed to demonstrate:

- How to set up parallel processes
- How to bind processes together using a synchronisation channel
- How to couple processes together for the exchange of data.

The examples include checks for successful allocation of processes and channels. It is recommended that similar checks be included in all application programs to ensure correct error handling when process or channel space fails to be allocated.

The `p = p;` statement at the start of the code for each process disables the generation of a compiler warning message and has no operational effect in the code. For further details see section 4.7.1.

Example 1 – Unsynchronised parallel processes.

This example shows how to declare and run basic parallel processes in C. A time delay is introduced into one of the processes to demonstrate their independence from each other.

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

void hello(Process *p)
  {
    p = p;
    ProcWait(10000);
    printf("\nHello,\n");
  }

void world(Process *p)
  {
    p = p;
    printf("\nWorld\n");
  }

int main()
  {

    Process *p1, *p2;

    p1 = ProcAlloc(hello,0,0);
    if (p1 == NULL)
      abort();
    p2 = ProcAlloc(world,0,0);
    if (p2 == NULL)
      abort();

    ProcPar(p1,p2,NULL);

  }
```

Example 2 – Processes synchronised by a channel.

This example shows how the two processes in example 1 can synchronise their activity using channel communication. Using a channel to connect the two processes forces process **world** to wait until **hello** has completed its output, and makes the processes interdependent. No status polling is required because synchronization is implicit in the channel reference.

The *integer* channel functions are used for convenience. In this instance any pair of channel functions will do the job providing the communication protocols agree. The channel simply ties the two processes together, and communicates no real data.

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>

void hello(Process *p, Channel *ready)
  {
    p = p;
    ProcWait(10000);
    printf("\nHello,\n");
    ChanOutInt(ready, 1);
  }

void world(Process *p, Channel *ready)
  {
    p = p;
    ChanInInt(ready);
    printf("\nWorld\n");
  }

int main()
  {

    Process *p1, *p2;
    Channel *ready;


    ready = ChanAlloc();
    if (ready == NULL)
      abort();

    p1 = ProcAlloc(hello,0,1,ready);
    if (p1 == NULL)
      abort();
    p2 = ProcAlloc(world,0,1,ready);
    if (p2 == NULL)
      abort();

    ProcPar(p1,p2,NULL);

  }
```

Example 3 – Communicating data over a channel.

This example shows how two processes can both synchronise their behaviour and communicate data by the use of channels.

In the example the process **input** prompts the user for a name and passes it to a second process **display** which adds it to a predefined string and displays a personalized greeting.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>

void input(Process *p, Channel *chan)
  {
    char message[20];

    p = p;
    printf("\nPlease type your name (20 letters max): ");
    gets(message);
    ChanOut(chan, message, 20);
  }

void display(Process *p, Channel *chan)
  {
     char name[20];

     p = p;
     ChanIn(chan, name, 20);

     printf("\nHello %s\n", name);

  }

int main()
  {

    Process *p1, *p2;
    Channel *chan;

    chan = ChanAlloc();
    if (chan == NULL)
      abort();

    p1 = ProcAlloc(input,0,1,chan);
    if (p1 == NULL)
```

```
    abort();
  p2 = ProcAlloc(display,0,1,chan);
  if (p2 == NULL)
    abort();

  ProcPar(p1,p2,NULL);

}
```

5 Introduction to the ANSI C compiler

This chapter provides an introduction to the ANSI C compiler and describes its main sequential and concurrent features. It describes the meaning of transputer types and transputer classes and how they can be used to generate common code for groups of processors. The chapter also provides a short introduction to the Runtime Library and outlines the support provided for low level and mixed language programming.

5.1 Introduction

The ANSI C compiler is an ANSI standard C compiler with concurrency extensions to support parallel programming for transputers and transputer networks. The ANSI C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler written by Drs. Arthur Norman and Alan Mycroft.

The ANSI C compiler implements fully the X3.159-1989 ANSI standard for the C programming language. This standard is expected to be ratified as ISO standard ISO 9899 and to become the internationally recognised standard for the C programming language. The standard specifies the content and defines the interpretation of programs written in C, establishing standards of reliability, and maintainability and enhancing portability of programs between systems.

ANSI C supports a standard model of parallel processing based on *processes* which interact via *channels*. The system is based on a set of predefined data types and structures and a set of library functions for creating processes and synchronising data transfer down channels. Semaphores are also supported using data types and library functions.

5.2 Source and object code

The C compiler takes as input an ANSI standard C source file and compiles it into an intermediate object file in standard format for linking with the linker tool **`ilink`**. The compiled object code is compatible with code generated by other INMOS compiler toolsets that generate TCOFF object code. Once linked, the code is converted to an executable program binary file using the configurer and collector tools.

Command line options control the target processor type and other facilities such as the degree of compiler checking, the format of error displays, and the output

file format. An option is available to disable code generation and output assembly code to a file.

5.2.1 Object code format

The compiler generates intermediate object code (TCOFF) that can be processed by other tools in the toolset and when linked can be mixed with code written using other compatible INMOS toolsets.

Unlike the standard UNIX C compiler `icc` does not automatically link the program or generate an executable module. Code generated by `icc` must be linked transputer bootstrap information added before it can be run on a transputer. Multitransputer programs must also include a configuration stage which describes how the software is to be placed on the transputer network.

The steps involved in generating transputer executable code are described in chapter 3. Briefly, the compiler produces an object code file in the standard TCOFF format. This compiled object file must then be linked with all other code modules and libraries using the linker tool. Once the code is linked the final stages which generate loadable transputer code differ for single and multitransputer programs.

For multitransputer programs the linked unit is configured on a network with other linked units by the configurer tool, and subsequently processed by the collector tool. For single transputer programs the collector is used directly to append bootstrap code to the linked unit. In both cases the file generated is one that can be loaded onto transputer hardware using the `iserver` tool.

5.3 Transputer types and classes

This section describes the meaning of transputer types and classes and how selection of the target processor affects the compilation and linking stages of program development. The section describes how to compile and link code targetted at a single processor type and then describes how to compile and link programs so that they can be executed on different processor types.

5.3.1 Single transputer type

For those users who have a single transputer or indeed a network of transputers all of the same type, the compilation and linking stages of program development are very straightforward. Simply compile and link all your modules for the required processor.

The compiler and linker both support command line options to select the following processor types:

16-bit processors	T212, M212, T222, T225
32-bit processors	T400, T414, T425, T800, T801, T805

Example: to compile and link for a T800:

```
icc hello -t800
ilink hello.tco -t800 -f startup.lnk          (UNIX)

icc hello /t800
ilink hello.tco /t800 /f startup.lnk    (MS-DOS and
VMS)
```

The default target processor for both the compiler and linker is a T414, so if you are using this processor type the steps are even simpler:

```
icc hello
ilink hello.tco -f startup.lnk                (UNIX)

icc hello
ilink hello.tco /f startup.lnk       (MS-DOS and VMS)
```

5.3.2 Creating a program which can run on a range of transputers

The compiler and linker use the concept of *transputer class* to enable programs to be developed which may be run on different transputer types without the need to recompile.

A transputer class identifies an instruction set which is common to all the processors in that class. When a program is compiled and linked for a transputer class it may be run on any member of that class.

Note: Code created for a transputer class will often be less efficient than code created for a specific processor type. Therefore, creating code for a transputer class is discouraged in situations where program efficiency is a primary concern; it should only be performed where there is a genuine need to produce code which will run on a range of transputers or to reduce the size of a support library, where program efficiency is not a major concern.

Table 5.1 lists all the transputer classes which the compiler and linker support and indicates which processors the program can be run on.

In order to develop a program which will run on different processor types, perform the following steps:

Transputer class	Processors which class can be run on
T2	T212, M212, T222, T225
T3	T225
T4	T414, T400, T425
T5	T400, T425
T8	T800, T801, T805
T9	T801, T805
TA	T400, T414, T425, T800, T801, T805
TB	T400, T414, T425

Table 5.1 Transputer classes and target processor

1 Identify the processors on which the program is to run.

2 Using table 5.1 select the class which may be run on all the target processors.

3 Compile and link all the program modules for this class.

For example to create a program which will run on both a T400 and a T425, compile and link for transputer class T5:

```
icc hello -t5
ilink hello.tco -t5 -f startup.lnk                (UNIX)

icc hello /t5
ilink hello.tco /t5 /f startup.lnk  (MS-DOS and VMS)
```

Alternatively to create a program which will run on a T400, T425 or a T800, compile and link for transputer class TA.

```
icc hello -ta
ilink hello.tco -ta -f startup.lnk                (UNIX)

icc hello /ta
ilink hello.tco /ta / startup.lnk   (MS-DOS and VMS)
```

Programs compiled for the T212, M212 or T222 transputers, which make up class T2, can be run on a T225 (class T3) because a T225 has a similar but larger instruction set than class T2 transputers. Similarly code compiled for a T414 (class T4) may be run on a T400 or T425, which form class T5. The T400 and T425 have additional instructions to those of the T414. Likewise, code compiled for a T800 (class T8) may be run on a T801 or T805, which form class

T9. Again the T801 and T805 have additional instructions to those of the T800.

5.3.3 Object file containing code compiled for different targets

This section describes how object code compiled for one target processor or transputer class can be linked with code compiled for different transputer types or classes.

The ability to do this provides the user with greater flexibility in the use of program modules:

- An individual module can be compiled once e.g. for class T4, and then linked with separate programs to run on different processor types e.g. T414 and T425.
- When the user is preparing a library for use by programs intended to run on different processor types, a single copy of code compiled for a transputer class can be inserted instead of multiple copies for specific transputers.

When linking a collection of compiled units together into a single linked unit, the user must select a specific transputer type or transputer class on which the linked unit is to run. As before, this determines the set of transputer types on which the code will run. When linking for a particular type or class, the linker will accept compilation units compiled for a compatible class. Table 5.2 shows which transputer classes the linker will accept when linking for a particular class.

Link class	Transputer classes which may be linked
T2	T2
T3	T3, T2
T4	T4, TB, TA
T5	T5, T4, TB, TA
T8	T8
T9	T9, T8
TB	TB, TA
TA	TA

Table 5.2 Linking transputer classes

For example if the target processors are a T400 and a T425 the user may compile for classes T5 and TB and link the code for for class T5.

Code for a different transputer class can be included in the final linked unit, as long as :

- it uses the instruction set or a subset, of the instruction set of the link class.

- the calling conventions are the same.

Classes T8 and T9 cannot be linked with class TA. This is a change from the previous issue of the toolset. The reason why these classes cannot be linked together is explained in section 5.3.4, which gives details of the differences between the instruction sets, as additional information.

A library can be made, consisting of the same modules compiled for different transputer types or classes. The user then needs only to specify the library file to the linker, and the linker will choose a version of a required routine which is suitable for the system being linked.

The linker uses the rules given in table 5.2 to determine whether a compiled module, found in a library, is suitable for linking with the current system. So, for example, to create a library which may be linked with any transputer class or specific transputer type, all routines could be compiled for classes T2, TA and T8.

If there are a number of possible versions of a module in a library the best one (i.e. the most specific for the system being linked) is chosen.

5.3.4 Classes/instruction sets – additional information

The instruction sets of the transputer classes differ in the following ways:

- Classes T2 and T3 support 16-bit transputers whereas all the other transputer classes support 32-bit transputers.

- Class T3 is the same as class T2 except that T3 has some extra instructions to support CRC and bit operations, special debugging functions and includes the *dup* instruction.

- Class T5 is the same as class T4 except that T5 has extra instructions to perform CRC, 2D block moves, bit operations, special debugging functions and also includes the *dup* instruction.

- Class T9 is the same as class T8 except T9 has additional debugging instructions.

- The T800, T801 and T805 processors use an on-chip floating point processor to perform REAL arithmetic. Thus a large number of floating point instructions are available for these transputers and for their associated classes T8 and T9. These instructions are listed in section B.7.

- For the T414, T400 and T425 processors i.e. transputer classes T4 and T5 the implementation of REAL arithmetic is in the software. These transputers make use of a small number of floating point support instructions listed in section B.6.

- The instruction set of class TA only uses instructions which are common to the T400, T414, T425, T800, T801 and T805 transputers. Therefore it does not use the floating point instructions, the floating point support instructions or the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.

- The instruction set of class TB only uses instructions which are common to the T400, T414 and T425 processors. Therefore it uses the floating point support instructions, but does not use the extra instructions to perform CRC, 2D block moves or special debugging or bit operations and it does not use the *dup* instruction.

When considering the similarities and differences in the instruction sets of different transputer classes it helps to divide them into the three separate structures as shown in figure 5.1.

By comparison with Table 5.2 it can be seen that a module may only be linked with modules compiled for a transputer class which belongs to the same structure.

Classes T2 and T3 which form the first structure are targetted at 16-bit transputers so it is obvious that they cannot be linked with the other classes which are all targetted at 32-bit transputers.

The reason why classes T8 and T9 cannot be linked with classes TA, TB, T5 or T4 is because floating point results from functions are returned in a floating point register for T8 and T9 code and in an integer register for all other 32-bit processors. Even if your code does not perform real arithmetic, linking code compiled for a T9 or T8 with code compiled for any of the other classes is not permitted.

To summarise, compiling code for the transputer classes TA and TB enables it to be run on a large number of transputer types, however, the code may not be as efficient as code compiled for one of the other transputer classes or for a specific transputer type. For example compiling code for class T5 enables the

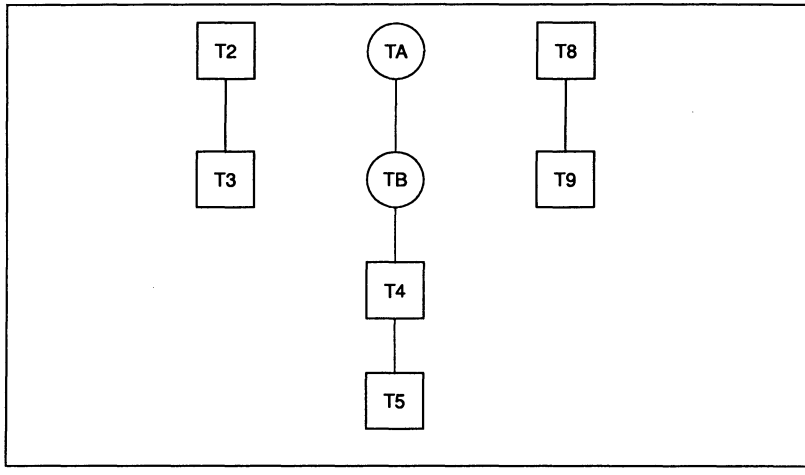


Figure 5.1 Structures for mixing transputer types and classes

CRC and 2D block move instructions to be used, whereas these instructions are not available to code compiled for classes TA and TB.

5.4 Error modes

The compiler generates object code in a single error mode known as UNIVERSAL. Other types of error modes can be generated by other INMOS compilers, for example, the occam 2 compiler `oc`, and may be encountered in mixed language programs. The other two common modes are: HALT, which halts the transputer when the program generates a runtime error; and STOP, which stops the errant process but allows the rest of the program to continue.

The linker requires all modules to be in the same error mode. Command line options are provided for setting all program modules to the same mode and should be used in mixed language programs containing code compiled in HALT or STOP modes, so that all C modules are converted to the correct mode.

5.5 Preprocessor directives

The ANSI C compiler incorporates an ANSI standard C preprocessor that allows source file inclusion, conditional code, conditional and unconditional definitions, and implementation dependent pragmas. The following directives are supported:

`#include`	`#else`	`#define`
`#elif`	`#undef`	`#endif`
`#if`	`#line`	`#ifdef`
`#pragma`	`#ifndef`	`#error`

Details of the compiler directives can be found in chapter 11.

5.5.1 Include files

Include files can contain declarations, definitions, or code. Header files for the runtime library are imported using the `#include` directive.

The search paths for files imported with the `#include` directive are similar to those for the toolset as a whole (see section A.3), but differ in some important respects. Two forms of syntax can be used to specify the filename, one of which allows the search path to be extended by directories specified on the command line. For more details see section 11.3.1.

5.5.2 Pragmas

The `#pragma` directive allows some compiler operations to be activated or deactivated in specific sections of code. Pragmas are defined for setting or overriding compiler options, particularly those concerned with code checking, for defining the size of linker code patches, and for allowing code written in other languages to be called from C.

The pragmas provided with `icc` are listed below.

`IMS_on`	`IMS_off`	`IMS_nolink`
`IMS_linkage`	`IMS_modpatchsize`	`IMS_codepatchsize`
`IMS_translate`		

Details of pragma syntax and options can be found in section 11.3.11.

5.5.3 Compiler messages

Compilation errors are displayed to the user by compiler messages. The format used for compiler messages is the same as the standard format used by all of the tools for error messages and is described in section A.6.1.

Compiler messages are generated at four different severity levels: *Warning*; *Error*; *Serious*; and *Fatal*.

Errors which indicate non-compliance with ANSI C are generated at severity levels *Error* and *Serious*. It may be possible to compile non-compliant code by using the '-E*' series of compiler options which disable certain ANSI checks. Severities greater than *Warning* prevent the generation of object code.

The display format for compiler messages starts with the error severity, gives the filename and line number where the error was found (if appropriate), and then gives a short reason for the error.

Compiler options can be used to suppress certain types of error messages. If these options are specified on the command line the corresponding error messages are not generated.

A complete list of compiler messages and their meanings can be found in chapter 11.

5.6 Runtime library

The ANSI C runtime library contains the full set of ANSI functions, a set of concurrency functions, and some other miscellaneous non-ANSI functions. The library is supplied as compiled code for all transputer types and classes; the correct library code is selected at link time, based on the transputer target specified at compilation. A reduced form of the library is supplied for programs which require no server-based communications, for example in embedded systems. The correct library is linked into the program using separate linker command files for the full and reduced libraries.

Access to the functions is via header files which are included at the start of a program. These contain function declarations, constants, and common variables. The ANSI library uses the standard ANSI set of header files, the concurrency library is split over three files dealing with process control, channels, and semaphores, and the miscellaneous functions are divided into several files containing related groups.

The library is installed on the system as two files, one containing the full set of functions, and the other containing all functions except those that require access to the host file server. The two libraries are known as the *full* and *reduced* libraries respectively.

The library files are modular, with a fine granularity. Each module contains either a single function, or a few related functions, so that only the minimum amount of code is loaded. The library is indexed for quick look-up by the compiler and linker.

5.6.1 Reduced library

Code which does not communicate with the host file server, that is, does not use any of the server-based functions, can be linked with the reduced runtime library. Using the reduced library ensures that code that is not required, such as the code that ensures proper closedown of the i/o system, is not loaded with the program.

The reduced library contains all the functions (including concurrency functions) that are in the full library, but omits those which require the host file server. This includes common ANSI functions such as **printf** and **getenv** and i/o dependent functions such as **host_info**.

The reduced library can be used to limit the size of code in systems where memory space is limited, such as embedded systems. It can also be used to generate code for remote nodes in a transputer network, that is, those that have no direct dialogue with the host. (Nodes can still communicate with each other using the channel functions, which are included in the reduced library.)

A few functions from the standard i/o library, not true i/o functions, are available in the reduced library. These are the functions **sprintf**, **sscanf**, and **vsprintf**, which are used to format and deformat strings. The three functions are declared in the header file **stdiored.h**.

5.7 Low level programming

icc incorporates support for low level programming in the form of a machine code insertion facility and some predefined names which can be used to obtain a limited amount of low level information about compiled code.

5.7.1 Assembly code support

The compiler provides support for in line transputer assembly code in C programs. Sequences of transputer instructions can be embedded in C code using the **__asm** construct.

__asm can be useful for implementing low level operations such as controlling peripheral devices, and for optimising the performance of critical sections of code. It is not intended for the wholesale inclusion of large blocks of assembly code and should not be used for this purpose.

Details of how to use the assembly code insertion facility, with examples illustrating commonly performed operations, can be found in chapter 4 '*Language extensions*' of the accompanying Reference Manual.

5.7.2 Compiler predefines

The two predefined names `_lsb` and `_params` can be used as variables to determine the position of a file's static data and a function's parameter block respectively. For further details see section 11.5.3.

5.8 Mixed language programming

Code written using other INMOS language compilers that generate TCOFF standard file format (such as the occam 2 compiler `oc`) can be incorporated into C programs, with certain restrictions, by defining them as external functions and then linking them in the normal way. The compiler pragma `IMS_nolink` is then used to compile the function in the C program without a static link parameter.

Mixed language programs can be constructed easily using the configuration system. Individual linked units written in different languages can be placed on any transputer in a network; to the configurer all linked units are the same and can be mixed in any combination. The method can also be used for mixing code on the same, or a standalone, processor; in this case the processor is simply treated as a single-node network and configured in just the same way.

Chapter 9 explains how to create mixed language programs using the configurer, how to import occam code into C programs, and how to call C functions from occam.

6 Configuring transputer programs

This chapter describes the configuration language in detail and shows how it is used to map software programs onto transputer networks. The chapter includes examples of some simple configuration descriptions for single and two-processor networks.

6.1 Introduction

Transputer programs can be configured to execute on any physical arrangement of transputers. The assignment of independent but communicating program units to a specific transputer network is known as *configuration.*

Configuration is achieved by first writing a *configuration description* in the network configuration language. The configuration description is then processed by the configurer tool to generate a configuration data file, and then by the collector tool to generate a transputer loadable file, which can be loaded directly onto the network.

Within the configuration description software and hardware networks are defined independently and married together using a mapping description. The mapping description assigns software modules to specific processors and places channel inputs and outputs on transputer links. The software modules referenced in the configuration description must be linked units.

6.2 Configuration model

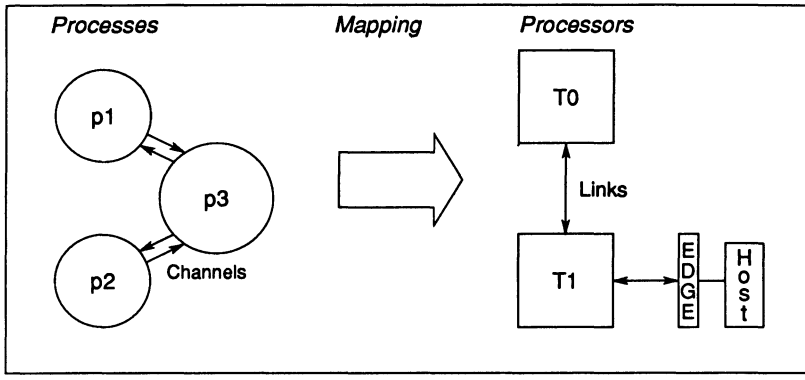
The configuration model consists of a software network consisting of *processes* joined to a hardware network of *processors* by a *mapping description.*

The software network is a description of the way in which processes interact; processes are defined with specific interface parameters and connected to each other using input and output channels defined in the configuration language. Configuration channels, like channels in source programs, are unidirectional, point-to-point connections. The number of channels is unlimited. Connections to the outside world are made via channels declared outside the processes known as channel edges. Channel edges can be connected to channels defined within a process to enable connections to be made to other software networks.

The hardware network describes the processors on the network and the physical link connections between them. The number of links available on each processor

is determined by the processor type. The hardware network interfaces to the outside world via a special configuration type called an **edge**.

The main elements of the configuration model are illustrated below.



Software and hardware networks are built up from generic network types called **nodes**. Definitions of software nodes (**processes**) and hardware nodes (**processors**) that form the basic elements of software and hardware networks are configuration defaults which are read from a file of predefinitions when the configurer is invoked.

6.3 Configuration language

The network configuration language is a special purpose language that allows linked object code to be connected to other linked units and placed on any physical arrangement of transputers. The language has been designed to be compatible with several language toolsets and allows linked code to be mixed on the same network. The main features of the language are listed below.

- The language is C-like. Declarations and expressions use C notation.
- Software and hardware networks are described in a simple declarative way using a common syntax.
- Replication and conditional statements make it easy to describe regular networks, and to define irregularities within them.
- New nodes types can be created from existing node types.
- Identifiers have global scope (except replication counters).

- Arrays can be declared of processes, processors, channels, and edges.
- Source files can be included.
- Comments can be inserted at any point.
- All statements except **if** and **rep** must be terminated with a semicolon.

A formal description of the language can be found in appendix C. The following sections describe the main features of the language and explain each of the language statements.

6.3.1 Identifiers

Identifiers represent elements used in the configuration, for example processors, processes, and channels. Most identifiers will be associated with a type, although untyped constant identifiers are also permitted.

Symbolic names can consist of any alphabetic character, any decimal digit, or the underscore character but must begin with a letter or underscore. All characters in the name are significant and case sensitive.

6.3.2 Types

Six base types are defined within the language:

node	**input**	**output**	**edge**
connection	*numeric value*		

The predefinitions **process** and **processor** derived from the **node** type are contained in a file that is read by the configurer at startup.

numeric value can have the following subtypes:

char	(character)	**int**	(32-bit integer)
float	(32-bit IEEE real)	**double**	(64-bit IEEE real)

The **char** subtype represents the integer value of a character's ASCII code. The default subtype for a numeric constant if none is implied, is **int**.

Node types are associated with a number of *attributes*. The **element** attribute is common to software and hardware nodes and depending on its value defines attributes which are specific for software and hardware nodes.

6.3.3 Constants

Numeric and character constants are defined using the **val** statement. A subtype can be specified but if it is not it will be deduced from the expression. For example:

```
val gridsize 4;     - integer
val x_coord 2.0f;   - single length real
```

Integers can be expressed in decimal, octal, or hexadecimal. Suffixes **K** and **M** can be used to indicate 'Kilo' (2^{10}) and 'Mega' (2^{20}) values for example, when defining processor memory size.

Character constants must be enclosed within single quotes and string constants within double quotes. Standard C escape sequences can be used to specify control characters such as Tab and EOL ('end-of-line'). For example:

```
val c 'c';                  - character constant
val greeting "Hello\n";     - string constant
```

Note: Any string constant that is to be passed to a C program must be explicitly terminated by the NULL character escape sequence **\0**.

For a full list of escape sequences supported in the configuration language see section C.4.3.

Constant arrays can be defined by enclosing the sequence of values in braces. Multidimensional constant arrays are permitted. For example:

```
val pow2 {1, 2, 4, 8, 16, 32, 64, 128};

val powers {{1, 1, 1}, {2, 4, 8}, {3, 6, 9}};
```

6.3.4 Booleans

The boolean constants **TRUE** and **FALSE** are predefined as integer constants with values one and zero respectively. In conditional statements any non-zero expression counts as **TRUE**.

6.3.5 Expressions and arithmetic

Expressions follow the syntax of the C language. Standard C operator precedence determines the order of evaluation, and brackets can be used to override the normal ordering. Operators supported are as follows:

Unary: `! - + ~`

Binary: `+ - * / % & | ^ && || << >> < <= > >= == !=`

Ternary: `?:`

All integer arithmetic is carried out to 32 bit precision and also passed as 32 bit integer values regardless of the processor type, that is, it is independent of the word length of the processor.

Strings and arrays can be tested for equality in the same way as integer expressions using the == and != operators.

6.3.6 Arrays

Arrays can be defined of any base or user-defined type. The size and dimensions are specified after the symbol name using the square bracket convention. Subscripts are numbered from zero. Values are stored in row order.

All elements in constant arrays must be of the same type and for multidimensional constant arrays the dimension size of all the subarrays must be consistent.

Arrays are commonly used to define the basic elements of the hardware and software networks. For example:

```
processor grid[4];

process slave[4];
```

Elements of arrays can be referenced by specifying the subscript either after the array name or after the array declaration. For example:

```
val y x[i];

val x {1, 2, 3}[i];
```

6.3.7 Conditional statement

The **if...else** statement controls the execution of the succeeding statement. The syntax of the statement is as follows:

if *exp statement* [**else** *statement*];

where: exp is any valid expression.

statement can be a single statement or a group of statements.

if is commonly used to exclude part of a network from a replicated declaration. For example:

```
T414(memory=1M) grid[4];

edge freelink[4];

rep i = 0 to 3
  {
  connect grid[i].link[2] to grid[i].link[3];

  if (i == 0)
    connect grid[i].link[1] to host;
  else
    connect grid[i].link[1] to freelink[i];
  }
```

if can also be useful to selectively place processes on specific types of processors, for example:

```
if remote.memory >= 2M
  place master on remote;
else
  place master on root;
```

This places the **master** process on processor **remote** only if the processor is equipped with at least 2 Megabytes of memory.

6.3.8 Replication

The **rep** statement replicates the succeeding statement or group of statements. **rep** is a counted loop in which the control bounds are integers or integer expressions.

The **rep** statement has two syntactic forms in which the number of replications is specified either by a range of values or by an initial value followed by a count:

rep *index* **=** *exp* **to** *exp statement*

rep *index* **=** *exp* **for** *exp statement*

For example:

```
rep i = 0 to 9
  { ...
  }

rep i = 0 for 10
  { ...
  }
```

If the range or count is zero the succeeding statement or group of statements is not executed.

Replication is commonly used to define regular networks such as grids, rings, and hyper-cubes and to place processes on them. It can be used for both hardware and software networks.

The following example of the use of **rep** connects four T414 transputers in a square-array and places the same process on each. The processors are connected to their neighbours via links 2 and 3; links 0 and 1 processor are left unconnected:

```
T414(memory=1M) grid[4];

rep i = 0 to 3
  connect grid[i].link[2] to grid[i].link[3];

process slave[4]

rep i = 0 for 4
  place slave[i] on grid[i];
```

6.3.9 Predefined functions

The function **size**(*array*) is predefined. **size** returns the number of elements in its array argument. If the argument is not an array then **size** returns the value 1 (one).

6.4 Network definition

Software and hardware networks are defined using a common syntax based on the declaration of nodes and their interconnection by language statements.

6.4.1 Nodes

Nodes are a generic network type from which the software node type **process** and hardware node type **processor** are derived. Although not a formal part of language syntax, process and processor types are predefined in a configuration defaults file input by the configurer and can be used as though they are defined in the language. Definitions of all the predefined types can be found in section C.3.2.

Nodes are associated with a number of *attributes*, the exact number and nature of which depends on the common attribute **element**. Elements of type **processor** imply the presence of the **type** and **memory** attributes, and elements of type **process** imply the existence of a set of runtime process attributes such as interface parameters, priority of execution, memory requirements, and program memory segment ordering.

Node attributes can be accessed in expressions using the dot convention and can be used to control the configuration. For example:

```
if (remote.memory >= 2M)
  place p on remote;
else
  place p on root;
```

Process and processor attributes are described in more detail below.

6.4.2 New node types

Refinements of existing node types can be created by using the **define** statement to specify nodes with specific attributes. For example, the predefined node types **process** and **processor** are defined in the following way:

```
define node(element="processor") processor;

define node(element="process") process;
```

Node types can be used to define other types. For example, the base type processor can be refined into a TRAM definition in the following way:

```
define processor(type = "T414") T414;

define T414(memory = 1M) B403;
```

New software node types can be defined in the same way. For example:

```
define process(stacksize = 10K,
               interface (int count,
                          input command,
                          output result)) workpackage;
```

Once defined, new types can be used to declare variables in the same way as base types. For example:

```
T414 worker;

B403 root;

workpackage slave[4];
```

6.4.3 Connections

Nodes are connected by the **connect** statement which can be used to join software channels (unidirectional), transputer links (bidirectional), or network edges (bidirectional). The statement has two equivalent syntactic forms:

connect *item* **to** *item* [**by** *connection*]

connect *item* , *item* [**by** *connection*]

Examples of the use of **connect** can be found in succeeding sections.

Prohibited connections

Connecting processes to processors, inputs to inputs, and outputs to outputs (except channel to edge connections) is prohibited and generates a configurer error.

Connections can also be named for later use in the configuration, using the **connection** type.

6.5 Software network description

The software network is composed of nodes of the predefined type **process** connected by input and output channels. The software description consists of a series of process declarations along with the program statements that connect processes together and define the network's interface with the outside world. A separate statement is used to assign compiled and linked modules to the software processes.

A typical process declaration would be as follows:

```
process(stacksize=2K,
        interface(int count, input in)) p;
```

6.5.1 Process attributes

Each process possesses a set of attributes some of which must be given values in the process declaration; others are optional with built-in defaults. The attributes of a process are as follows:

stacksize	The size of stack required for the process in bytes. Must be specified.
heapsize	The size of heap required for the process in bytes. Must be specified.
interface	The list of parameters to the process. Should be specified if external communications are required.
priority	The execution priority for the process. Priority can be **HIGH** or **LOW**. Optional.
order	The ordering of program segments in memory. Segments which can be ordered are **code**, **stack**, **static**, **heap**, and **vector**. Optional.

6.5.2 Stack and heap size

For C programs **stacksize** and **heapsize** are mandatory. The sizes of the program segments **code**, **vector**, and **static** are fixed by the compiler and linker.

6.5.3 Interface

The **interface** attribute is mandatory and defines the way in which the process interacts with the outside world or other processes via a set of parameters. The parameters can be read by a source program using the runtime library function **get_param**.

Parameters can be input channels, output channels, simple datatypes, and arrays of these. Permitted datatypes for the parameters are **int**, **char**, **float**, and **double**. Strings are defined as arrays of characters and may be initialised by a quoted string constant.

Each input channel can only be connected to an output channel on another process and vice versa for output channels. The rules for connecting channels to software network edges is described in section 6.5.8.

Values can be defined for interface parameters either by assigning a value in the process declaration or by a separate statement. For example:

```
process (interface (int count = 10,
                    input command,
                    output result)) task;

\* count defined at declaration as 10 *\

task (count = 100);

\* count redefined as 100; this value for count
   remains in force until redefined again *\
```

Values given to parameters may also be derived from a replicator count using an expression including the index variable.

Array parameters

When assigning parameters which are numeric arrays it is not possible to assign individual elements of the array but only the complete array. For example:

```
x(y = {0,1,2,3})
```

`get_param` function

A special library function `get_param` is provided to receive the process interface parameters within the C program. The function returns pointers to the parameters and is used to retrieve them from the configuration code.

Details of the function's operation can be found under the function description in the accompanying Reference Manual.

Host server channels

For C programs which use the host file server the first scalar input channel and the first scalar output channel in the list of parameters must be the host server channels. It should be noted that it is the programmer's responsibility to ensure the channels are constructed correctly and declared in the configuration. The

server channels, like other interface parameters, can be accessed by calling **get_param**.

For programs linked with the reduced library there is no access to the server and therefore the server channels are not required. In this case all channels declared in the interface definition are those defined for use in the program.

Note: The first arrays of input and output channels defined in the interface definition are passed into the main entry point as arrays of pointers to channels. However, they are still accessible through **get_param** and this is the recommended way of retrieving them.

6.5.4 Execution priority

Runtime priority for the process can be set high or low by specifying **priority=HIGH** or **priority=LOW**. The default is **LOW** priority.

6.5.5 Segment ordering

The order in which the four program segments are placed in transputer memory can be changed by specifying an ordering priority for each or any of the four code segments. The default is no segment ordering.

The syntax for the **order** attribute is as follows:

order ({*segment* = *value*})

where: segment is one of: **code stack vector static heap**

value is the ordering priority and can take any integer value. Positive values indicate placement higher in memory and imply lower speed access; negative values indicate lower memory placement and imply higher speed access. The lower the placement, the greater the chance that code will be placed in on-chip RAM, which has the fastest access.

If no order is specified a default segment ordering is applied. Details of the ordering can be found in section 12.3.7.

6.5.6 Defining new process types

Specific process types can be defined and used later in the program with different actual parameters. For example, the following code defines a process type **filter** which is later used to declare three filter processes with different values for the **cutoff** parameter. The processes are configured to form a pipeline starting and finishing at the host. The connection statements linking interface channel variables to host channels are shown for completeness. The host channels are assumed to have been defined earlier in the program.

```
process (stacksize = 200,
         interface (input in,
         output out, int cutoff)) filter;

filter x, y, z;

x(cutoff = 10);
y(cutoff = 20);
z(cutoff = 10);

connect x.in to from_host;
connect y.in to x.out;
connect z.in to y.out;
connect z.out to to_host;
```

Attributes can also be activated for specific instances of a type by specifying them within the declaration. In the following example a heap size is defined for a single instance of the **worker** type; no other instances are affected and all other attributes are unchanged:

```
worker (heapsize = 50K) ant;
```

Extra attributes can also be supplied in process definitions or in attribute assignment statements. For example:

```
define worker (heapsize = 20K) small_worker;
small_worker drone;

worker bee;
bee (heapsize = 200K);
```

6.5.7 Input and output channels

Processes which cooperate with each other and exchange data are connected by channels of types **input** or **output**. These channels are equivalent to channels in source code programs.

To send or receive data processes must declare input and output channels. The sending process must declare an input channel and the receiving process an output channel. The two channels are then connected by a **connect** statement.

In the following example a host monitor process **host_process** sends data via the output channel **from_host** to the application process **p**, which receives it on the input channel **in**.

```
process(stacksize = 200,
        interface (int count, input in)) p;

process(stacksize = 200,
        interface (output from_host)) host_process;

connect p.in to host_process.from_host;
```

6.5.8 Edge connections

The software network can be connected to the outside world by channel edges. Channel edges are input and output channels declared within the software description and connected to input and output channels of a process. A process can then import or export data from the software network. Edges are commonly used for interfacing i/o processes to the host server.

Unlike channels between processes, connections between edges and process channels must be of the same polarity, that is, an input edge must be connected to an input channel and an output edge to an output channel. This preserves the direction of the channel parameter. For example, given the process **p** in the above example, the following code creates an interface between **p** and the host server:

```
input from_server;                  /* input edge */
output to_server;                   /* output edge */

connect p.in to from_server;        /* input path */
connect p.out to to_server;         /* output path */
```

6.5.9 Assigning code to processes

Code is assigned to specific processes by means of the **use** statement. This associates a specific object module with a process. The module must be a *linked* unit. In the following example the same linked module **filter.lku** is

assigned to each of three processes:

```
filter x, y, z;

use "filter.lku" for x;
use "filter.lku" for y;
use "filter.lku" for z;
```

Linked units can also be associated with process *types*. This allows the same code to be assigned to several processes in a single statement. For example:

```
filter x, y, z;

use "filter.lku" for filter;
```

6.6 Hardware network description

Hardware networks consist of nodes of type **processor** connected by processor links. The hardware description contains declarations of processors on the network along with the connect statements that join them by their processor links.

Processors have two user-definable attributes:

type	The processor type. INMOS standard transputer types are predefined.
memory	The amount of memory available to the processor.

All attributes can be specified when the processor is declared.

Links are special attributes of processors that predefined within the language. Once the **type** is defined all processors declared using that name acquire the appropriate number of links. Link attributes cannot be changed.

Links can be connected to links on other processors or mapped onto software channels.

A typical processor declaration would be:

```
processor(type = "T414", memory = 1M) root;
```

6.6.1 Processor links

The number of links on each processor type is predefined within the configurer via the processor **type** attribute. The value of this attribute is defined for all INMOS transputer types listed in the standard include file **setconf.inc**.

Links are referenced using the dot notation and can be treated as arrays. For example, the **size()** function can be used to determine the number of links on a processor:

```
size(T414.link)
```

6.6.2 Defining new processor types

Processor types can be defined for later use in a program. In the following example the processor type **T800** is first defined and then used to define a further processor type called **B405** which is a T800 with a set amount of memory. This definition corresponds to the INMOS *iq* systems IMS B405 TRAM product.

```
define processor(type = "T800") T800;

define T800(memory = 8M) B405;
```

Certain processor types are predefined in the configurer by the automatic inclusion of the **setconf.inc** file at startup. The file provides definitions of all transputer types manufactured by INMOS along with other predefinitions.

Predefined types can be used as though they are part of the language and do not need to be referenced by an **include** statement.

The definitions are listed in section C.3.2.

6.6.3 Links

Processors are connected to each other by processor links. The number of links is defined by the processor type. Link numbers begin at zero.

Links can only be connected to one other link (or network edge, see below). Links can be left unconnected.

Links are specified using the dot convention as in C structures. They can also be subscripted as though they are arrays. For example:

```
connect root.link[2] to transputer1.link[0]
```

6.6.4 Edges

Edges are hardware network variables which bring transputer links out of the network for connection to the outside world, that is, to external devices or other networks. They are directly analogous to channel edges in software networks. Edges have the same characteristics as processor links. Edges can only be connected to other transputer links.

In the following example an edge is declared which allows a processor in a hardware network to input data from an A-to-D convertor:

```
T414 data_handler;

edge a_to_d;

connect data_handler.link[1] to a_to_d;
```

A special edge called **host** is predefined in configurer defaults file and can be used without defining it in the program. In networks that will be loaded from a host system, there must be one, and only one, processor link connected to **host**.

Arrays of edges can be useful constructions. In the following example an array of edges is declared for a series of sampling lines and then connected to three links of a processor which logs the data from each line. The remaining link is used to boot the processor.

```
edge samplers[3];

rep i = 0 to 2
  connect data_logger.link[i] to samplers[i];
```

6.7 Mapping description

The mapping description defines how processes and channels declared in the software description should be assigned to processors and links defined in the hardware description. Assignment is performed in both cases by the **place** statement:

place *process* **on** *processor;*

place *channel* **on** *link;*

6.7.1 Placement of channels

The configurer *automatically* places software channels on links using the placement of processes processors as a guide. Explicit placement of channels on links is only required where links are used for special purposes, for example, connection to a device, or where an application uses input and output channels separately, as in software implementations of high-speed links.

The configurer also performs automatic placement of one end of a connection if the other end is explicitly placed.

Predefined `connection` names can also be used to place named channels on named links. For example:

```
connect root.link[0] to host by root_link0_host;

connect master.ts to to_server by master_ts_to_server;
connect master.fs to from_server by master_fs_from_server;

place master_ts_to_server on root_link0_host;
place master_fs_from_server on root_link0_host;
```

Note that the order of automatic channel placements is defined by the order of the elements in the `connect` statements. In the above example the placements would be as follows:

```
place master.ts on root.link[0];
place master.fs on root.link[0];

place to_server on host;
place from_server on host;
```

6.8 Software network example

In the following example the software network consists of an i/o process `host_process` and a worker process `task` connected by the channels `in` and `out`. Two processors `root` and `processor1` joined by a single link connection form the hardware network. The root processor is already connected to the host via link zero. The mapping description places each process on one of the processors and assigns channels between the processes to the appropriate links:

```
connect host_process.out, task.in;
connect host_process.in, task.out;
```

```
connect root.link[1] to processor1.link[0];

place host_process on root;
place task on processor1;
```

6.9 Terminating configured processes

Configured processes (processes that have been configured on a processor by **icconf**) cannot use **exit** to terminate the program. In the case of configured processes **exit** merely terminates the process from which it is called; it does not affect the server and other processes will continue to run.

To terminate the server from a configured process use **exit_terminate**, which shuts down the server and terminates the program. Details of the function can be found in chapter 2 of the accompanying Reference Manual.

Configured processes which use the *reduced* library cannot terminate the program (even by using **exit_terminate**) because they have no link with the server. In these cases a call to **exit_terminate** has the same effect as **exit**.

6.10 Checking the configuration

Configurations may be checked against the hardware on a transputer board using a network check program such as **ispy**. The **ispy** program supplied as part of the board support software for INMOS *i*q systems products. These products are available separately from your local INMOS distributor.

6.11 Configuration examples

Note: The examples presented here are simply intended to illustrate the syntax of configuration language statements and how they are used to form a configuration description. They are not intended to be tutorial examples and are *not* provided on the toolset **examples** directory.

Further examples illustrating how to use the configuration language and configure software on various network topologies can be found on the **config** examples subdirectory. This subdirectory contains the program configuration source files and a number of Makefiles and batch files to assist with program building. A READ ME file provides a summary of the directory contents, describes the prerequisite hardware, and gives instructions on how to build the programs.

Note: A thorough knowledge of the way the configuration language defines software and hardware networks and links them by mapping statements is a pre-requisite to understanding the configuration model. Readers are recommended to study the examples at length and be thoroughly familiar with the language before attempting to write complex configurations.

Example 1 – Single process configured on one transputer.

The following example shows how to configure a program consisting of a single process on one transputer. The single process contains all the code in the program including that required for host communication.

```
/* Hardware description:
   declare processor memory size;
   connect link 1 to host edge          */

T800 (memory = 1M) root;

connect host to root.link[1];

/* Software description: declare channels;
   declare process and interface params;
   connect interface to inputs and outputs  */

input from_server;
output to_server;

process (stacksize = 8K, heapsize = 50K,
         interface (input command,
                    output reply)) job;

connect from_server to job.command;
connect to_server to job.reply;

/* Mapping description:
   define object file;
   place process on processor;
   place channels on edge "host"  */

use "job.lku" for job;

place job on root;

place from_server on host;
place to_server on host;
```

Example 2 – Two processes configured on a two-processor network.

The program consists of two processes. One process acts as the interface to the host (the i/o process) and the other performs a complex numerical calculation.

The hardware consists of a T425 and a T800 transputer connected together by a single link. The T425 acts as the root transputer. The i/o process is to be executed on the T425 and the numerical process on the T800.

```
/* Hardware description: */

T425 (memory = 1M) root;
T800 (memory = 2M) worker;

connect root.link[1] to host;
connect root.link[2] to worker.link[1];

/* Software description: */

input from_server;    /* input edge */
output to_server;     /* output edge */

process (stacksize = 8K, heapsize = 50K,
         interface (input command,
                    output reply,
                    output feed,
                    input response)) controller;

connect from_server to controller.command;
connect to_server to controller.reply;

process (stacksize = 16K, heapsize = 512K,
         interface (input feed,
                    output response)) task;

connect controller.feed to task.feed;
connect task.response to controller.response;

/* Mapping description: */

use "control.lku" for controller;
use "compute.lku" for task;

place controller on root;
place task on worker;

place from_server on host;
place to_server on host;
```

6.12 Configuration language summary

Numeric types	
int	Integer type.
char	Character type.
float	Single length floating point type (IEEE 754).
double	Double length floating point type (IEEE 754).
Configuration types	
node	A point in a software or hardware network. Has the general attribute **element** (**process** or **processor**) and other specific attributes for software and hardware nodes.
connection	A defined connection type between links or channels.
edge	Declares a network edge.
input	Declares a software process *input* channel or edge.
output	Declares a software process *output* channel or edge.
Language constructs	
if	**if** *exp statement* **else** *statement* Simple conditional construct. *exp* can be any valid integer expression and *statement* can be the single succeeding statement or a group of statements. **else** is optional.
rep	**rep** *index* = *exp* **to** *exp statement* **rep** *index* = *exp* **for** *exp statement* Simple replication construct. Can be controlled by a range or a count.
connect	**connect** *item* **to** *item* **by** *connection;* **connect** *item , item* **by** *connection;* Joins channels to channels, links to links, channels to software edges, and links to hardware edges. **by** is optional.
place	**place** *process* **on** *processor;* **place** *channel* **on** *link;* Assigns a software process to a processor, or a channel to a link.
use	**use** *filename* **for** *process;* Assigns a linked unit to a process.
#include	**#include** *filename* Includes another source file.

Definitions	
`val`	`val` *identifier exp;* Defines a numeric constant. The type is deduced from the type of the expression.
`define`	`define` *type* (*attributes*) *identifier;* Defines a node type. A list of attributes is optional.
Operators	
	Unary: `+ - ! ~` Binary: `+ - * / % & | ^ << >> && || < > <= >= == !=` Ternary: `?:`
Functions	
`size`	Returns the size of an array.

Node types	
`process PROCESS`	Software process node type.
`processor PROCESSOR`	Hardware processor node type.
`T212 t212 T222 t222 T225 t225 M212 m212`	IMS T2 series.
`T400 t400 T414 t414 T425 t425`	IMS T4 series.
`T800 t800 T801 t801 T805 t805`	IMS T8 series.
Constants	
`HIGH, high`	The integer constant 0 (zero). Used to indicate a high priority process.
`LOW, low`	The integer constant 1 (one). Used to indicate a low priority process.
`TRUE, true`	The integer constant 1 (one).
`FALSE, false`	The integer constant 0 (zero).
Edges	
`host`	The host link or channel.

7 Loading transputer programs

This chapter explains how to load programs onto single transputers and transputer networks. It briefly describes the format of loadable programs and introduces the program loading tools **iserver** and **iskip**. The chapter goes on to explain how to load programs for debugging and ends with an example of skip loading.

7.1 Introduction

Transputer programs are loaded onto transputer boards with the **iserver** tool which installs code on each processor using processor and distribution information embedded in the executable file. The executable file consists of code to which bootstrap information has been added to make the program self-booting on the transputer. Self-booting executable code is also known as **bootable** code.

Bootable files are generated by **icollect** from configuration data files (network programs) or linked units (single transputer programs). Bootable files are generated with the default extension **.btl** (for loading onto boot from link boards), or **.btr** (for loading onto boot from ROM boards).

7.2 Tools for loading

Two tools are provided to load programs onto transputers and transputer networks:

- **iserver** – the file server and loader tool.

  iserver loads the bootable file onto the single transputer or transputer network and activates the host file server that provides communication with the host.

- **iskip** – the skip loading tool.

  iskip allows a program to be loaded over the root transputer onto an external network. The tool is used prior to invoking **iserver** to start up a special route-through process on the root transputer that transfers data between the the network and the host system.

Skip loading is useful for the post-mortem debugging of programs that use the root transputer. The root transputer in the network is omitted from the logical network and the program is loaded onto the first processor *after* the root transputer, leaving it free to run the debugger. This avoids having to debug the code from a memory dump file.

Programs loaded using **iskip** always require one extra processor on the network in addition to those required to run the program. For example, a program written for a single transputer requires at least two processors, one to act as the root transputer and one to run the program.

7.3 The loading mechanism

In single transputer programs code is loaded onto the first processor on the network and the program code is then loaded down the host link byte by byte. If **iskip** has been used the program is loaded onto the second processor on the physical network. In multitransputer programs the process is repeated for all processors on the network until all the code is loaded.

When the code is copied into the transputer's memory the process boots automatically and the program continues to run until an error occurs or the server is terminated by pressing the ISERVER interrupt key, usually CTRL-C or CTRL-BREAK.

7.3.1 Breakpoint debugging

Programs are loaded for breakpoint debugging using the **idebug** command. When invoked in breakpoint mode this command incorporates a skip load and **iserver** is not required. Because it uses a skip load, breakpoint debugging requires at least two processors on the network.

For more information about breakpoint debugging and details of the command syntax see section 15.3.4.

7.4 Boards and subnetworks

There are two basic types of transputer evaluation board: those that boot from link and those that boot from ROM.

Boot from link boards form the majority of transputer boards in general use. They are loaded down the link that connects the root transputer to the host using the **iserver** tool. Programs intended to run on boot from link boards must consist of bootable code.

Examples of boot from link boards supplied by INMOS are the IMS B008 PC motherboard and the IMS B014 and IMS B016 VMEbus standard interface boards.

Boot from ROM TRAMs boards are intended for standalone applications such as embedded systems.

Examples of boot from ROM products are the INMOS *i***q** systems IMS B418 Flash ROM TRAM and the IMS B016 VME board operating in boot-from-ROM mode.

7.4.1 Subsystem wiring

Subsystem wiring is the way in which boards are connected together, and determines the manner in which transputer subnetworks are controlled.

Three signals are used to control transputers mounted in a system, namely **Reset**, **Analyse**, and **Error**. Together these are known as the *System Services*. All INMOS transputer boards use a common scheme for propagating these signals to other subnetworks. The scheme is as follows.

Each transputer board has three ports for communicating system services from one board to another. These are **Up**, **Down**, and **Subsystem**. Up is the *input* port, used to control the board from an external source; Down and Subsystem are both output ports and are used to propagate the Up signal to other boards or subnetworks.

The Down and Subsystem ports work in the following ways:

Down propagates the Up signal unchanged to the next board or subnetwork. This allows multiple boards to be chained together by connecting successive Up and Down ports and the whole network can be controlled by a single signal.

Subsystem transfers control to the board, allowing subnetworks downstream of the board to be independently reset, analysed, and their error flags read, under the control of the root transputer on that board.

7.4.2 Connecting subnetworks

Multiple transputer systems can either be controlled by the host computer or by a *master* transputer controlled by the host computer.

In a typical multitransputer system the root transputer's Up port is connected to the host computer so that it can control the loading of programs and monitor

errors on the network. The first processor in the subnetwork is connected to either Down or Subsystem depending on the application, and other processors on the network are chained together via their Up and Down ports.

In a simple application requiring multiple transputers, the subnetwork would normally be connected to Down on the root transputer. This would allow the host computer to reset the whole network in a single operation and to monitor the error signal on any transputer in the network.

A more complicated application may require several programs to be loaded onto the subnetwork under the control of the root transputer. Here the subnetwork would be connected to Subsystem so that the root transputer could repeatedly reset and re-load the subnetwork. Any errors in the subnetwork would be detected by the root transputer through its Subsystem port, and the error would not be propagated through the Up port to the host computer. Reset and Analyse signals are propagated through to the Subsystem port, but the error signal is not relayed back.

7.5 Loading programs for debugging

Special debugger and server options must be used for the debugging of programs running on transputer boards. The options vary with the subsystem wiring, the board type, and whether or not the program uses the root transputer. The effects of subsystem wiring are described above; the effects of board type and program mode are described in the following sections.

Commands to use for various combinations of subsystem wiring, board type, and program mode, are listed in Table 15.2.

7.5.1 Board types

Some early INMOS boards of the B004 type, unlike later TRAM-based boards, do not propagate Reset through to the Subsystem port. On these boards the '**SA**' iserver option must be supplied on the debugger command line to reset the network.

7.5.2 Use of the root transputer

The use made of the root transputer by the program changes the procedures you must use in post-mortem debugging. This is because the debugger program executes on the root transputer and any application code becomes overwritten when the tool is invoked.

Two procedures can be used to load and debug code running on the root transputer:

1 Programs can be loaded in the normal way using **iserver** and the program image in the root transputer's memory saved to a file. The code running on the root transputer is then debugged from the dump file. Code running on the rest of the network is debugged in the normal way by reading the transputer memory directly down the transputer links.

   The dump file is created by invoking **idump**. The debugger is subsequently invoked using the debugger '**R**' option that directs it to read the dump file.

   Note: On boards that contain only one transputer this method *must* be used.

2 Programs can be loaded over the top of the root transputer by invoking the **iskip** tool before **iserver**. This leaves the root transputer free to run the debugger. The program can then be debugged down the root transputer link in the normal way.

   If **iskip** is used an extra processor is required over and above those required to run the application program.

Programs configured for a subnetwork that does not include the root transputer can be loaded with **iserver** and debugged down the root transputer link using the debugger '**T**' option.

Details of the procedures to use for loading and debugging all types of transputer programs can be found in section 15.2.

7.5.3 Analyse and Reset

Care must be taken that **Analyse** or **Reset** are only asserted once on a network that is to be debugged, or incorrect data will be obtained. To ensure this the debugger should be invoked using the standard command sequences given in Table 15.2.

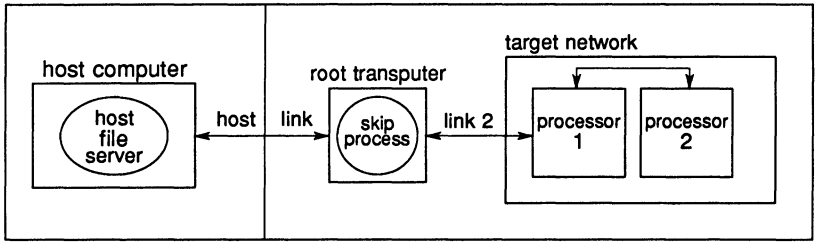
7.6 Example skip load

This section shows how to load a program into a network over the root transputer using the **iskip** tool.

7.6.1 Target network

The program to be loaded is configured for a target network consisting of two T800 processors mounted on a B008 motherboard. The target network is connected to a T414 processor in slot zero acting as the root transputer, and the two T800 processors are connected by a single link.

The target network and its connections are shown schematically below.



7.6.2 Loading the program

The file **twinprog.btl** contains the bootable program.

To prepare the board for running the program on the target network, invoke **iskip** using one of the following commands:

```
iskip 2 -r -e
iskip 2 /r /e
```

This sets up the system to direct the program to the target network over the top of the root transputer and starts the route-through process on the root transputer. Options '**r**' and '**e**' respectively reset the target network and direct the host file server to monitor the halt-on-error flag.

The program can then be loaded using one of the following commands:

```
iserver -ss -se -sc twinprog.btl          (UNIX)
iserver /ss /se /sc twinprog.btl          (MS-DOS and VMS)
```

7.6.3 Clearing the network

On transputer boards error flags can be cleared using a network check program such as **ispy**. (Error flags can become set when the board is powered up).

The `ispy` program is provided as part of the board support software for INMOS *iq* systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared. This method is described in section 15.3.4.

8 Debugging transputer programs

This chapter describes the facilities of the toolset debugger **idebug** and shows how they can be used to debug transputer programs in a systematic way. It explains how the debugger can be used in two ways (post-mortem and interactive) to analyse transputer programs and describes the two levels of debugging (symbolic and Monitor page). The chapter includes examples to illustrate the debugging facilities and demonstrate debugging techniques, and ends with a list of points to note when using the debugger.

8.1 Introduction

The network debugger **idebug** is a comprehensive debugging tool for transputer programs. It can be run in post-mortem mode to determine the cause of failure in a halted program, or in interactive mode to execute a program stepwise by setting breakpoints in the code. In either mode programs can be debugged from source code using the symbolic functions or from the machine code using the Monitor page commands. The two environments can be invoked from each other at will.

Post-mortem debugging allows programs to be examined for the cause of failure after halting the transputer on error. The debugger locates the errant process in the program either by direct examination of the program image in transputer memory or by reading memory dump files. Processes running in parallel with the errant process can be examined anywhere on the network.

Breakpoint debugging allows programs to be executed in a stepwise manner under interactive control. Breakpoints can be set within the code to cause the program to pause for the inspection of variables, channels, and processes; variables can be modified and the program continued with the new values.

The debugger can also be invoked on a dummy network to examine the static features of a program. The dummy network simulates the contents of memory locations and registers, and can also be used to explore the features of the debugger without running a real program.

8.1.1 Debugging with isim

The transputer simulator tool **isim** can also be used to debug transputer programs from a low level environment. Using a similar environment to the debugger Monitor page transputer memory can be examined, breakpoints set, and

programs executed by single stepping.

The debugging facilities of the simulator are briefly described in this chapter (section 8.12). Details of how to use the simulator tool can be found in chapter 24.

8.2 Programs that can be debugged

The debugger can analyse programs running on transputers that are either directly attached to a host through a server program, or connected to the host via a root transputer. The debugger runs on the root transputer and networks to be debugged must incorporate a 32-bit transputer at the root. If breakpoint debugging is used the transputer network must contain at least two processors, because the root transputer is dedicated to running the breakpoint debugger.

8.3 Compiling programs for debugging

Programs to be debugged should be compiled with full debugging information enabled.

8.3.1 Symbolic debug information

Full debugging information, necessary for debugging, is selected by specifying the compiler 'G' option when the program is compiled.

By default `icc` generates object files containing minimal symbolic debug information. This is in order that object modules, especially those intended to form libraries, are kept as small as possible. Minimal debug information enables the debugger to backtrace out of a library function to a module compiled with full debug information.

Note: The object code produced with minimal debug information contains certain optimisations that are absent in code generated with full debugging information enabled. As a consequence the object code produced may be different.

8.3.2 Error modes

C programs are compiled in toolset error mode UNIVERSAL, which enables them to be mixed freely with modules compiled in other modes (HALT or STOP) using other INMOS language toolsets. The programmer need take no special action to ensure complete compatibility with other INMOS compilers.

The error mode of a mixed language program may be changed at link time; the

linker default is to generate HALT code, which is the recommended mode for debugging.

Further information about link error modes can be found in section 20.4.2.

8.4 Debugging configured programs

Configured programs (programs created from a configuration description by **icconf**), must be processed using the configurer '**G**' option to generate debugger compatible information.

8.5 Post mortem debugging

Post-mortem debugging refers to the analysis of stopped programs, that is, programs that have failed to run correctly and set the transputer error flag. Programs that are to be debugged in this mode should be configured in HALT mode so that the processor halts when the flag is set, and they should be loaded by **iserver** so that the error flag is monitored, by specifying the '**SE**' option.

Post-mortem debugging can also be used to debug programs that have been explicitly interrupted by the host system BREAK key. To interrupt a program, for example when a program 'hangs', press the BREAK key, which stops the server but not the program, and then invoke **idump** to take a snapshot of the running program. Invoking **idump** stops the program by sending an Analyse signal to the transputer in order to take a snapshot of its current activity.

Automatic error checking, for example of array indices, is not provided in C and this makes it difficult to cause a C program to HALT when an error occurs. This restricts somewhat the usefulness of post-mortem debugging in C, but can be used if programs are halted explicitly by using the debugging support functions (see section 8.11) or the library functions **abort** and **assert** (see below).

Breakpoint debugging with its associated debug support functions is a more flexible approach and is the recommended method for debugging C programs if possible.

Using `abort` to halt a program

The **abort** function can be enabled to halt the processor by calling the auxiliary function **set_abort_action**. This enables a backtrace to be performed to the point in a program where the error occurred, without the need to modify all of the assert statements in a program.

The technique is illustrated in the following example:

```
/*************************************
*
*  Debugger example:  abort.c
*
*  Example of forcing a C program to HALT the
*  processor for post-mortem analysis regardless
*  of the error mode it has been configured in.
*
*  Use of the debug support functions is encouraged
*  as an alternative (see debugger example file debug.c
*  for details).
*
*************************************/


#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <assert.h>


int
main (void)

{
    /*  0 will cause assert () to fail assertion test  */
    int     x = 0;

    printf ("Program started\n");

    /*  override normal abort action  */
    set_abort_action (ABORT_HALT);

    printf ("Program being halted by assert ()\n");
    assert (x);

    printf ("Program being halted by abort ()\n");
    abort ();
}
```

8.5.1 Program loading

Programs which run on the root transputer, or which use the root transputer to run part of a multiprocessor program, must be debugged from an memory image of the transputer. This is necessary because the debugger executes on the root transputer and overwrites the code in the transputer's memory.

The memory dump is performed using the **idump** tool after the program has failed and before the debugger is invoked with the 'R' option. Details of how to invoke the **idump** tool can be found in chapter 16.

Alternatively the program can be skip loaded onto the next processor on the network, avoiding the root transputer. This requires one extra processor on the network over and above the number needed to run the program. Skip loading is described in chapter 25.

If only one transputer is available, for example on single-transputer boards, the memory dump method *must* be used. If more than one transputer is available skip loading is the recommended method since it is a quicker operation.

8.6 Breakpoint debugging

Breakpoint debugging allows programs to executed under interactive control using breakpoints set in the code. Breakpoints can be set on any line of source. Symbolic and Monitor page facilities can be used to examine code, inspect variables, jump down channels to other processes or processors, and determine the state of the network. Special symbolic functions and Monitor page commands, only available in breakpoint mode, support the modification of variables and memory locations and the restarting of programs from the breakpoint or from other points in the code.

8.6.1 Runtime kernel

The breakpoint debugger places a special runtime kernel on each processor in addition to the application bootable code. This kernel provides a virtual communication network to enable the debugger to transparently share transputer links with the application in addition to providing a breakpoint handler to deal with breakpoints, errors, inspection of processor state etc. The scheme is illustrated in Figure 8.1.

Note: The debugging kernel places the transputer into Halt-On-Error mode regardless of the error mode of the program. This means that during breakpoint debugging a transputer will always HALT when an error occurs.

The runtime kernel requires a certain amount of memory on each processor, the exact amount differing slightly between processor types. The size of the kernel on each transputer type is given in Table 8.1.

Apart from the extra memory required, the kernel is transparent to the application program if processes on different processors communicate with each other in the

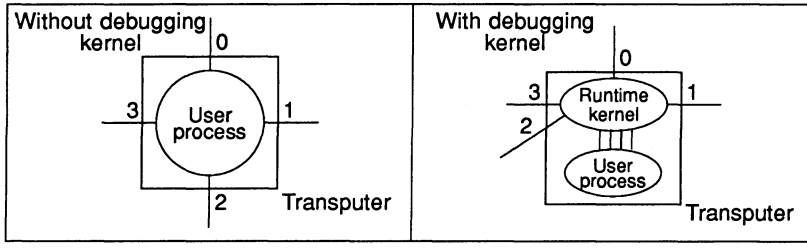


Figure 8.1 Debugger runtime kernel

Processor	Kernel size	H/W support
M212	10K	No
T212	10K	No
T222	10K	No
T225	12K	Yes
T414	12K	No
T800	12K	No
T400	14K	Yes
T425	14K	Yes
T801	14K	Yes
T805	14K	Yes

Table 8.1 Runtime kernel size and processor breakpoint support

normal way using channels supplied by the configurer (maximum of four input and four output per processor).

Note: To allow breakpoint debugging to function correctly a program must not place channels explicitly onto processor link addresses. Programs that do so may introduce conflict with the runtime kernel, which also uses the external links. Programs currently coded in this way should be recoded to pass in external channels, otherwise breakpoint debugging may not be used.

8.6.2 Hardware breakpoint support

Certain transputers have built-in instructions for breakpointing (see Table 8.1). For those processors without hardware breakpoint support, breakpoints should not be set within high priority processes because the mechanism used to implement breakpoints causes high priority processes to lock the processor and disable all communications to the processor via the runtime kernel.

The effect on the network of setting such a breakpoint will depend on the position

of the processor in the network hierarchy but in any event should be avoided. The debugger is unable to check the validity of breakpoints and it is the programmer's responsibility to ensure correct operation on processors without direct hardware breakpoint support.

8.6.3 Compiling the program

All modules in the program must be compiled in the same or a compatible mode. Modes are checked at link time and if incompatibilities are found the link is aborted.

8.6.4 Loading the program

Breakpoint debugging does not require special loading or memory dump procedures because the program is automatically skip loaded by **idebug**. However, breakpoint debugging does require one extra processor on the network because the root processor is dedicated to running the breakpoint debugger program.

8.6.5 Clearing error flags

If either **iserver** or **idebug** detect that the error flag is set immediately a program starts executing it is likely that the network consists of more processors than you are currently using and that one or more of the unused processors has its error flag set. (Error flags can become set when transputer boards are powered up).

On transputer boards error flags can be cleared by running a network check program such as **ispy**. This ensures a clean network on which to load the program.

The **ispy** program is provided as part of the board support software for INMOS *i*q systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program to clear the network is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared. This method is described in section 15.3.4.

8.6.6 Breakpoint functions and commands

Several symbolic debugging functions and Monitor page commands are only available in breakpoint mode. The commands available are summarised below.

Symbolic functions	
TOGGLE BREAK	Set/clear breakpoint.
RESUME	Execute from breakpoint.
CONTINUE FROM	Execute from current line.
MODIFY	Modify variable.

Monitor page commands	
B	Breakpoint menu.
J	Execute program.
S	Show debug messages.
U	Update register display.
W	Write to memory.

8.6.7 Breakpoints

Breakpoints can be set, cleared, and listed using Monitor page commands, and set/cleared using symbolic functions.

Breakpoints can be set at any point in a process running on any processor. At each breakpoint (or on error) the process pauses and the source code may be displayed.

Note: When a process is paused at a breakpoint or error other parallel processes in the program continue to run.

Breakpoints can be set at code entry points, or on any line of source code. Variables within scope at the breakpoint can be modified and the process restarted. Breakpoints can also be set at the Monitor page but care should be taken not to set breakpoints at addresses that do not correspond to the start of a source code statement, otherwise the behaviour is undefined.

Setting breakpoints at symbolic level is the recommended method.

8.7 Program termination

Program termination is signalled to the debugger by the termination of **`iserver`**. This is performed automatically by the C runtime system. If the program contains independently executing processes which do not require communication with the server the debugger may be resumed to interact with these processes.

To run or debug the program again it must be reloaded onto the transputer using **`iserver`**, or **`idebug`** in breakpoint mode.

8.8 Symbolic facilities

Symbolic debugging is debugging at source code level using the symbols defined in the program for variables, constants, and channels. Features provided in symbolic debugging include the examination of source code, the inspection of variables and channels, and the backtracing of procedure calls. A number of special breakpoint functions are available if the debugger is invoked in breakpoint mode.

Source level debugging is accessed through symbolic *functions* mapped to specific keyboard function keys. Keyboard layouts for specific terminal types can be found in the Delivery Manual that accompanies this release.

The main symbolic debugging activities and the functions that are used to access them are described in the following sections.

8.8.1 Locating to source code

Locating to the source code for a particular process is a crucial procedure in the debugging process on which other operations depend. For each required location the debugger must be given a memory address which it uses to locate to the source. When the required code is located, symbolic functions can be used to browse the code and inspect variables. Where the source code is unavailable, for example, libraries supplied as object code with minimal debug information; the line containing the library call is located to instead.

When first invoked in post-mortem mode the debugger is given the address of the last instruction executed, which it uses to automatically locate to the relevant source code. Subsequently for each new point to locate to in the code the debugger requires a new address which can be supplied by the programmer. A default address is available by pressing RETURN; normally the default address is the address of the previous location.

Addresses of important segments of code can be determined using the Monitor page commands that display lists of processes waiting on the run queues, the timer queue, and on the transputer links. Any address in memory can be specified using the Monitor page 'O' command.

Certain addresses are already known to the debugger and can be located to using symbolic functions without specifying the address or switching to Monitor page commands. Many of the common operations used during source code debugging can be performed directly with symbolic functions. They include relocating to the previous location and locating to the original error.

The symbolic functions that can be used directly for locating to known areas of

code are listed below.

TOP	Locate back to the error, or last source code location.
RELOCATE	Locate back to the last location line.

A strategy for debugging multiprocess programs by locating each process in turn is described later in this chapter in section 8.10.

8.8.2 Browsing source code

Several functions are available for browsing source files once they have been located. They include functions for navigating files, changing to included or new files, and string searching. The functions are listed below.

TOP OF FILE	Go to the first line.
BOTTOM OF FILE	Go to the last line.
GOTO LINE	Go to a specified line.
SEARCH	Search for a specified string.
ENTER FILE	Enter an included file (one incorporated by `#include`).
EXIT FILE	Exit to the enclosing file.
CHANGE FILE	Display a different file.

8.8.3 Inspecting variables

The values of constants, variables, parameters, arrays, and channels can be inspected at any point in the code. A special inspect function for channels only allows the debugger to locate to the process at the end of the channel. Symbols to be inspected must be in scope with the source line last located to.

Expressions can be used to inspect subsets of an array and to calculate values involving the inspected item. If the debugger is used in breakpoint mode variables can also be modified.

For C functions the debugger returns an address and for channels the debugger checks the channel's status and displays information about waiting processes. If no processes are waiting the channel is given as 'Empty'.

The two inspect functions are listed below.

INSPECT	Display the value and type of a source code symbol.
CHANNEL	Locate to the process waiting on a channel.

Jumping down channels

The CHANNEL function can be used to locate to a process waiting on a channel. This is known as 'jumping down' a channel and works for channels on the same processor (internal or *soft* channels) or channels assigned in the configuration to transputer links (external or *hard* channels which connect processes on different processors together). Debugging can then continue at the waiting process. If no process is waiting on a channel the channel is given as 'Empty'.

8.8.4 Tracing procedure calls

Two functions assist in the tracing of function calls. They can be used even if the source is not present, for example, libraries supplied as object code with minimal debug information, but in this case the line containing the function call is displayed rather than the library code itself. Where procedures are nested successive backtrace operations will locate to the original call. Variables and other symbols can be inspected at any stage. The two functions are listed below.

BACKTRACE	Locate to the procedure or function call.
RETRACE	Reverse the last BACKTRACE.

8.8.5 Modifying variables

The MODIFY function allows variables and constants to be changed in transputer memory and the program continued with the new values. It supports the same expression language as INSPECT. For further details see chapter 15.

8.8.6 Breakpointing

Symbolic functions are provided for setting and clearing breakpoints, for modifying the value of a variable, and for continuing the program.

TOGGLE BREAK	Set or clear a breakpoint on the current line.
MODIFY	Change the value of a variable in memory.
RESUME	Resume the program from the breakpoint.
CONTINUE FROM	Resume the program from the current line.

8.9 Monitor page

The debugger Monitor page is a low level debugging environment which gives direct access to machine level data. It allows memory to be viewed and disassembled and gives access to information about the processor's activity through the display of error flag status and pointers to process queues. Specific debugging operations are invoked by mainly single letter commands typed after the *Option* prompt.

8.9.1 Startup display

When first invoked in breakpoint mode, or in post-mortem mode with an invalid `Iptr` or `Wdesc` (see below), the debugger enters the Monitor page environment and displays information such as the addresses of instruction and workspace pointers, status of error flags, and information about the processor run queues. The memory map is also displayed.

If an `Iptr` or `Wdesc` is invalid at startup it is marked as invalid.

The Monitor page display differs slightly between post-mortem and breakpoint modes. In post-mortem mode the display includes the saved pointers for the low priority process if the processor was running at high priority when analysed; in breakpoint mode the display does not include these pointers but does include the contents of the A, B, and C registers, if known. At startup in breakpoint mode no machine pointers or register values are available (the program has not yet started) and so no values are displayed.

A typical post-mortem startup display is shown in figure 8.2.

```
Toolset Debugger : V2.00.00       Processor 0 "example" (T800)

  Processor State                        Memory map
Iptr           #80003B7A   Configuration code  : #80000070 - #8000014F (  224 )
Wdesc          #801FFE3D   Stack               : #80000150 - #8000076F ( 1568 )
Error          Set         Program code        : #80000770 - #80005A8F (   21K)
FPU Error      Clear       Configuration code  : #80005A90 - #80006293 ( 2052 )
Halt On Error  Set         Freespace           : #80006294 - #801FFFFF ( 2024K)
Fptr1 (low     Empty
Bptr1  queue)              Total memory usage  : 25236 bytes (25K)
Fptr0 (high    Empty
Bptr0  queue)              On-chip memory (4K) : #80000000 - #80000FFF
TPtr1 (timer   Empty       MemStart            : #80000070
Tptr0  queues) Empty
Clock1 (low)   #000234C5   Debugger has enough memory for 805 processors
Clock0 (high)  #008D3152



Error explicitly set, Last instruction was : seterr

Option (? for help) (A,C,D,E,F,G,H,I,K,L,M,N,O,P,Q,R,T,V,X,?) ?
```

Figure 8.2 Example post-mortem Monitor page display for a T800 processor

Items displayed on the startup page and their meanings are summarised in Table 8.2. Most of the data displayed is common to all transputer types. Where the display differs for specific processor types and debugging modes, this is indicated in the table.

Process pointers

Iptr points to the last instruction executed and **Wdesc** to the process workspace. Low priority **Iptr** and **Wdesc** are also displayed if the processor was running in high priority mode when it was halted. An asterisk placed next to either an **Iptr** or **Wdesc** indicates an invalid memory location for the process. 'Invalid' **Wdesc** indicates that no process was executing on the processor when it halted, which may occur in the presence of deadlock.

Practical note: If **Wdesc** contains the address of 'Memstart' it is likely that the **Analyse** signal has been asserted more than once on the network. This can occur on transputer boards where the subsystem signal is asserted on analyse, as on the IMS B004. For further guidance on the use of such boards refer to section 15.4.

Fptr and **Bptr** point to the process run queues, which hold information about processes awaiting execution. The suffix 1 indicates the high priority queue and 0 the low priority queue. If the front and back pointers are the same then only one process is waiting; if there are no processes waiting the pointers have no value and the queue is given as 'empty'.

Item displayed	Description
`Iptr`	Instruction pointer (address of the last instruction executed).
`Wdesc`	Workspace descriptor (pointer to process workspace).
`IptrIntSave`†	Saved low priority instruction pointer, if applicable.
`WdescIntSave`†	Saved low priority workspace descriptor, if applicable.
`A Register`‡	Contents of A register, if known.
`B Register`‡	Contents of B register, if known.
`C Register`‡	Contents of C register, if known.
`Error`	Status of transputer error flag.
`FPU Error`	Status of FPU error flag (T800 series only).
`Halt On Error`	Status of halt on error flag.
`Fptr1`	Front pointer to low priority process queue.
`Bptr1`	Back pointer to low priority process queue.
`Fptr0`	Front pointer to high priority process queue.
`Bptr0`	Back pointer to high priority process queue.
`TPtr1`	Pointer to low priority timer queue.
`TPtr0`	Pointer to high priority timer queue.
`Clock1`	Value of low priority transputer clock.
`Clock0`	Value of high priority transputer clock.

† Not available in breakpoint mode.
‡ Not available in post-mortem mode. Not known in breakpoint mode on processors with no hardware support for breakpointing.

Table 8.2 Items displayed at the Monitor page

`Tptr1` and `Tptr0` are pointers to the high and low priority timer queues respectively.

Registers

In breakpoint mode only, the contents of the transputer registers **Areg**, **Breg**, and **Creg** are displayed for those processors which have built in instructions for breakpoint handling. Values displayed are those which were current when the process stopped.

Error flags

Two flags are displayed for all processors: Error and Halt-on-error. The FPU Error flag is also displayed for transputers with an integral floating point unit (IMS T800 series).

Clocks

`Clock1` and `Clock0` display the values of the low and high speed transputer clocks when the process was stopped. In breakpoint mode the clock values (and queue pointers) can be updated using the Monitor page '`U`' command.

Memory map

The memory map display is included on the standard startup display, as though the Monitor page '`M`' option had been automatically invoked. Any or all of the following memory segments may be displayed, depending on the application program and its configuration:

Runtime kernel / Configuration code
Stack
Program code
Vectorspace
Static area
Heap area
Configuration code
Freespace

8.9.2 Monitor page commands

Most Monitor page options are single-letter commands that you type in at the Monitor page `Option` prompt. A few commands are mapped onto specific function keys. The commands that support breakpoint debugging are only available when the debugger is invoked in breakpoint mode.

The main Monitor page commands allow you to disassemble and display transputer memory, locate and debug processes, and examine the network processor by processor.

The main commands for common debugging operations are introduced in the following sections. Full details of all the commands can be found in chapter 15.

Examining memory

Specific segments of transputer memory can be displayed in hexadecimal, ASCII, or any high level language type, or disassembled into transputer instructions. The segment of memory to be displayed is specified by a starting address. A map of the transputer's memory can be displayed giving the positions of code and workspace. Commands for examining transputer memory are summarised below.

A	Display memory in ASCII.
D	Disassemble into transputer instructions.
H	Display memory in hexadecimal.
I	Display memory in selected data type.
M	Memory map.

Locating processes

Locating to code for specific processes is one of the major functions available through the Monitor page. They allow processes other than the stopped or current process to be located and examined anywhere on the network. Processes can be located on the current processor by examining run queues, and on other processors by jumping down transputer links.

Four commands are used, three to display waiting processes and one to jump to the selected code of a process displayed by the other three.

R	Display processes waiting on Run queues.
T	Display processes waiting on Timer queues.
L	Display processes waiting on Links.
G	Goto symbolic debugging for the selected process.

These commands can be used in a systematic way to trace all processes on a network and determine the cause of program failure. The method is explained in more detail in section 8.10.

Specifying processes

One command allows a specific process to be selected for symbolic debugging.

O	Specify a process for symbolic debugging.

The 'O' command is useful for going directly to symbolic debugging for a specific

process whose details you have already noted earlier in the debug session.

Selecting processes

The '**F**' command enables you to select a source file for symbolic display using the filename of the object module produced for it. This option enables symbolic locating (for setting breakpoints etc.) without needing to know **Iptr** and **Wdesc** process details (as the '**G**' and '**O**' options do).

Other processors

Two commands allow other processors on the network to be examined:

E	Go to next halted processor.
P	Go to specified processor.

Breakpoint commands

The following commands support breakpointing. To use the commands the debugger must be invoked with the '**B**' command line option.

B	Breakpoint menu.
J	Jump into and run application program.
S	Show debugging messages and prompts menu.
U	Update processor status display.
W	Write value to memory.

8.10 A method for debugging halted programs

Most transputer programs consist of several processes running in parallel, either on the same transputer or on a multitransputer network. The following technique is offered as a way of debugging halted programs using a systematic method based on the tracing of all processes in the network. The method can be used whether the program is running on a single transputer board or on a network of many processors.

8.10.1 Locating all processes

Processes are located by the debugger using the process **Wdesc** (Workspace Descriptor), which is a base pointer for the data and variables that make up the

process.

Each process running on a transputer exists in one of several states. In the systematic method each possibility is explored in turn until the errant process is found. The possible states for a process are:

- Not yet started.
- Running on the processor.
- Waiting on a processor execution queue (Run queue).
- Waiting on a timer execution queue (Timer queue).
- Waiting for communication from another process on the same processor.
- Waiting for communication on a transputer link (Link information).
- Already stopped or terminated.

Running on the processor

For the stopped process the debugger automatically locates to the area of source code where the error occurred.

Waiting on a run queue

Processes on the run queues can be located by first using the Monitor page 'R' command to display the list of waiting processes. A process can then be selected by pressing 'G' (for 'Goto process'), positioning the cursor on the desired process and pressing RETURN.

Pointers to the run queues are displayed on the Monitor page and can be used to determine the overall status of the queue. If pointer addresses are displayed there are processes waiting. If only a single process is waiting the front and back pointers have the same value. If no processes are waiting the queue is given as 'Empty'.

Waiting on a timer queue

Processes waiting for a specified time are placed on the high and low priority timer queues. These are similar to the run queues except that they are controlled by the transputer clock.

Processes on the timer queues can be located by using the Monitor page 'T'

command to display a list of processes and invoking the 'G' command to locate to the required process. Pointers to the timer queues are displayed on the Monitor page and can be used to determine overall queue status.

Waiting for communication on a link

Processes waiting for a hardware communication (input or output on a transputer link, or an input on the **Event** pin) can be located by using the Monitor page 'L' command to display a list of waiting processes, and invoking the 'G' command to locate to the process. Links where no processes are waiting are given as 'Empty'.

At most 9 processes can be waiting for a hardware communication, two for each of the four links and one for the **Event** pin. Pointers to these processes are held at special addresses at the bottom of the memory space and are not given on the Monitor page.

Waiting for communication on a channel

Processes waiting for a internal communication can be located from source level using the CHANNEL. If there are no processes waiting on a channel the channel is given as 'empty'.

Processes stopped, terminated or not started

If the running process and all the waiting processes have been found, not forgetting all those processes waiting on all the internal channels, then any processes still unaccounted for must either have finished or failed to start. These remaining processes cannot be located to because there are no `Wdescs` for them, and they must be accounted for by a process of elimination.

8.10.2 Locating functions

When a procedure is called, the workspace pointer is moved. If the debugger locates inside a function then only local variables, and variables declared globally, are in scope and available for inspection.

To inspect variables or channels not in scope within the function use the BACKTRACE key to locate to a position where the desired variable or channel is in scope. To relocate into the function again use the RETRACE key.

8.11 Library functions

Three functions are provided in the runtime library to assist with debugging. **debug_stop** and **debug_assert** are used to stop a process, the latter on a specified condition, and **debug_message** is used to insert debugging messages. The functions are accessed by including the header file **misc.h**.

Function	Action
debug_assert	Stops the process and alerts the debugger if the parameter condition evaluates FALSE (0).
debug_stop	Stops the process and alerts the debugger.
debug_message	Inserts debugging messages in the program.

Details of each of the functions can be found under the function descriptions in the accompanying Reference Manual.

debug_assert and **debug_stop** allow a process to be stopped at any point in the code, where it can then be debugged using the symbolic functions and Monitor page commands. **debug_stop** always stops the process whereas **debug_assert** only stops the process if the parameter condition evaluates to FALSE.

The following short example illustrates their use:

```
/*******************
*
*  Example of debug support functions when used with
*  and without the debugger.
*  (see also example file abort.c)
*
*******************/


#include <misc.h>
#include <stdio.h>


int
main (void)

{
  /*  0 will cause debug_assert to fail assertion test  */
  int     x = 0;

  printf ("Program started\n");
```

```
  debug_message("A debug message only within the
debugger");

  printf ("Program being halted by debug_assert ()\n");
  debug_assert (x);

  printf ("Program being halted by debug_stop ()\n");
  debug_stop ();
}
```

In this example if **x** is 1 **debug_assert** evaluates to TRUE and the program runs until it encounters **debug_stop**. If **x** is set to 0 (as in the example) **debug_assert** evaluates to FALSE and the process stops before it reaches **debug_stop**. Code stopped by **debug_assert** and **debug_stop** may be resumed from the line following the call of the debug function by using the CONTINUE FROM key.

debug_message is used to insert debugging messages into the code. Messages are relayed back to the terminal from any point in the program, even from code running on distant processors of a network. It can be used to monitor the activity of outlying processors which are not directly connected to the host. The display of debug messages at the terminal is controlled by an option on the Monitor page Breakpoint Menu.

8.11.1 Action when the debugger is not available

If the debugger is not available on the system the debug library functions have the following actions:

Function	Action
debug_assert	Stops the process (also stops the processor if configured in HALT mode) if the parameter evaluates to FALSE.
debug_stop	Stops the process (also stops the processor if configured in HALT mode).
debug_message	No action.

8.12 Debugging with isim

The T425 simulator **isim** provides a single processor interactive simulation of a program running on an IMS T425 transputer, running on a 2 Mbyte boot from link transputer board, and connected to a host computer through the host file server **iserver**. The interactive environment provides a machine level (non-symbolic)

environment similar to the debugger Monitor page for debugging programs and monitoring program execution.

The simulator allows any single processor program to be run and analysed without a transputer board.

All the component parts of a program to be simulated, must be compiled for the T425 transputer type (or compatible targets), linked together using **`ilink`** (including libraries), and made bootable using **`icollect`**.

Note: The simulator can only be used to simulate single transputer programs.

8.12.1 Command interface

The simulator has a single command interface which corresponds to the debugger Monitor page. Most commands are single letter commands and can be invoked with a single key press. For a list of commands see chapter 24.

8.12.2 Using the simulator

The simulator can be used in two ways:

- To debug programs by inspection of the transputer and memory, in the same way as with the debugger. Registers, memory, and machine state can be examined directly at the Monitor page.
- To monitor the execution of programs using machine level single step execution and the setting of break points at specific memory locations. Code can be executed by stepping single instructions.

8.12.3 Program execution monitoring

The simulator provides a number of functions that can be used interactively to monitor and control the behaviour of a program. These are:

- Breakpoints
- Single step execution of a program

A program can be stepped a single instruction at a time using the 'S' command.

Breakpoints

Breakpoints can be set, displayed, and cancelled using the 'B' command to display the Breakpoint Options Page.

Single step execution

A program can be stepped a single instruction at a time using the 'S' command.

8.12.4 Core dump file

`isim` may be used to produce a core dump file that can be read by the debugger (as if the code had been executed on a real transputer).

8.13 Debugging example

This example illustrates some of the post-mortem and breakpoint features of the debugger. The debugger is invoked in breakpoint mode.

8.13.1 The example program

The example program `facs.c` calculates the sum of the squares of the first *n* factorials, using a rather inefficient algorithm. It has been structured this way for clarity in process structure and to demonstrate parallel processing and debugging methods. The same program coded in occam is supplied with the occam 2 toolset.

The program incorporates five processes, each coded as a separate function. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result. The program is listed below.

```
/*****************************************
*
*  Debugger example:  facs.c
*
*  idebug (and parallel C) example based on similar program
*  in occam toolset.
*
*  Uses 5 processes to compute the sum of the squares of
the
*  first N factorials using a rather inefficient algorithm.
*
*  Plumbing:
*
*  - > feed -> facs -> square -> sum -> control <-> User
I/O
* |                                          |
*  ------------------------------------------
*
*****************************************/


#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>


const double    stop_real    = -1.0;
const int       stop_integer = -1;


void
ChanOutDouble (Channel *out, double value)

{
      ChanOut (out, (void *) &value, sizeof (value));
}


double
ChanInDouble (Channel *in)

{
      double  value;

      ChanIn (in, (void *) &value, sizeof (value));
      return value;
}
```

```
/*  compute factorial  */
double
factorial (int n)

{
      double  result;
      int     i;

      result = 1.0;
      for (i = 1; i <= n; ++i)   {
            result = result * i;
      }
      return result;
}


/*  source stream on ints  */
void
feed (Process *p, Channel *in, Channel *out)

{
      int     n, i;

      n = ChanInInt (in);
      for (i = 0; i < n; ++i)   {
            ChanOutInt (out, i);
      }
      ChanOutInt (out, stop_integer);
}


/*  generate stream of factorials  */
void
facs (Process *p, Channel *in, Channel *out)

{
      int     x;
      double  fac;

      x = ChanInInt (in);
      while (x != stop_integer)   {
            fac = factorial (x);
            ChanOutDouble (out, fac);
            x = ChanInInt (in);
      }
      ChanOutDouble (out, stop_real);
}
```

```
/*  generate stream of squares  */
void
square (Process *p, Channel *in, Channel *out)

{
      double  x, sq;

      x = ChanInDouble (in);
      while (x != stop_real)   {
              sq = x * x;
              ChanOutDouble (out, sq);
              x = ChanInDouble (in);
      }
      ChanOutDouble (out, stop_real);
}


/*  sum input  */
void
sum (Process *p, Channel *in, Channel *out)

{
      double  total, x;

      total = 0.0;
      x = ChanInDouble (in);
      while (x != stop_real)   {
              total = total + x;
              x = ChanInDouble (in);
      }
      ChanOutDouble (out, total);
}


/*  user interface and control  */
void
control (Process *p, Channel *in, Channel *out)

{
      double  value;
      int     n;

      printf ("Sum of the first n squares of
factorials\n");
      do   {
              printf ("Please type n : ");
      }   while (scanf ("%d", &n) != 1);
      printf ("n = %d\n", n);
```

```
        printf ("Calculating factorials ... ");

        ChanOutInt (out, n);
        value = ChanInDouble (in);

        printf ("\nThe result was : %g\n", value);
}


Channel *
Checked_ChanAlloc ()

{
        Channel *chan;

        if ((chan = ChanAlloc ()) == NULL)    {
                fprintf (stderr, "ChanAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return chan;
}


Process *
Checked_ProcAlloc (void (*func)(), int sp, int nparam,
                   Channel *c1, Channel *c2)

{
        Process *proc;

        proc = ProcAlloc (func, sp, nparam, c1, c2);
        if (proc == NULL)    {
                fprintf (stderr, "ProcAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return proc;
}


int
main (void)

{
        Channel *facs_to_square, *square_to_sum;
        Channel *sum_to_control, *feed_to_facs;
        Channel *control_to_feed;

        Process *p_feed, *p_facs, *p_square;
        Process *p_sum, *p_control;
```

```
    facs_to_square  = Checked_ChanAlloc ();
    square_to_sum   = Checked_ChanAlloc ();
    sum_to_control  = Checked_ChanAlloc ();
    feed_to_facs    = Checked_ChanAlloc ();
    control_to_feed = Checked_ChanAlloc ();

    p_feed = Checked_ProcAlloc (feed, 0, 2,
                    control_to_feed, feed_to_facs);
    p_facs = Checked_ProcAlloc (facs, 0, 2,
                    feed_to_facs, facs_to_square);
    p_square = Checked_ProcAlloc (square, 0, 2,
                    facs_to_square, square_to_sum);
    p_sum = Checked_ProcAlloc (sum, 0, 2,
                    square_to_sum, sum_to_control);
    p_control = Checked_ProcAlloc (control, 0, 2,
                    sum_to_control, control_to_feed);

    ProcPar (p_feed, p_facs, p_square, p_sum,
                                    p_control, NULL);

    exit (EXIT_SUCCESS);
}
```

8.13.2 Compiling and loading the `facs` program

The source of the program is provided on the toolset **debugger** examples subdirectory. It should be compiled for transputer class TA with debugging enabled, then linked with the appropriate library files and made bootable using **icollect** using the '**T**' option to create single transputer bootable code.

The example is intended for running on a B008 board wired *subs*. See section 15.4 if your system is different.

A typical sequence of commands for compiling, linking, and booting the program is shown below. The '**i**' option on the linker command line is optional but provides useful information on the progress of the linking operation.

Command sequences are shown for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

```
icc facs.c -g -ta -o facs.tax
ilink facs.tax -f startup.lnk -ta -o facs.cah -i
icollect facs.cah -t

icc facs.c /g /ta /o facs.tax
```

```
ilink facs.tax /f startup.lnk /ta /o facs.cah /i
icollect facs.cah /t
```

The program is loaded for breakpoint debugging by invoking **idebug** with the Breakpoint option using one of the following commands:

```
idebug -sr -si -b2 facs.btl -c t425

idebug /sr /si /b2 facs.btl /c t425
```

This command starts up the debugger and displays the Monitor page but does not start the program. The iserver '**si**' switch is optional.

Note: If your transputer is not a T425 you should change the **T425** option to the appropriate transputer type. You may also need to change the number specified after the '**b**' option to the number of the root transputer link where your network is connected.

See Table 15.2 for more details about the options to use if in doubt.

8.13.3 Setting initial breakpoints

Initial breakpoints can often be set by invoking the Monitor page '**B**' command and specifying a breakpoint at the start of **main()**. In this example we use a different method based on setting specific breakpoints in the source code before the program is started.

At the Monitor page select option '**F**' to display the source file. At the object module filename prompt specify the compiled object file **facs.tax**. The debugger uses debug information within the object module to select the source file.

The source file is displayed with the cursor positioned at the first function definition. At this point the program is still waiting to be started.

Set a breakpoint at the beginning of the **ChanOutDouble** function using [TOGGLE BREAK]. The debugger confirms the breakpoint is set. (Note that the breakpoint is set on the first executable line of the function.)

8.13.4 Starting the program

Return to the Monitor page using the [MONITOR] key and start the program by selecting the '**J**' option. Press [RETURN] at the 'Command line' prompt (no command line is required) and give a small positive number (e.g. 12) when the program prompts for input. The program runs until it reaches the breakpoint.

8.13.5 Entering the debugger

At the breakpoint the debugger requests confirmation to continue. Press any key except 'C' or 'c' to enter the symbolic debugging environment. The debugger locates to the breakpoint and displays the source code.

8.13.6 Inspecting variables

Variables and channels in **ChanOutDouble** can now be examined. For example, to examine the variable **value** press INSPECT and specify its name at the prompt. The debugger displays the value 1.0 and labels it as a **double**. Pressing INSPECT with the cursor positioned on **value** has the same effect.

Note that only variables in scope at the debugger's current location point can be inspected, although the rest of the file can be displayed with the cursor keys. The current location point is at the start of function **ChanOutDouble**.

8.13.7 Backtracing

ChanOutDouble is called from function **facs** to output the factorial it calculates for each integer received from **feed**. To confirm this press BACKTRACE and the debugger locates to the line in **facs** where **ChanOutDouble** is called.

8.13.8 Jumping down a channel

Within **facs** the variable **fac** is the first in a sequence of outputs on the channel **out**. To trace the destination process for **fac** first inspect the channel **out**, which is declared to be a channel pointer. Reinvoke INSPECT but specify ***out**, which dereferences the channel pointer. The debugger displays an **Iptr** and **Wdesc**, indicating that there is a low priority process waiting at the other end of the channel.

Now press CHANNEL and again specify ***out** to dereference the channel pointer. The debugger jumps down the channel connecting the two processes and locates to **ChanInDouble**. Now backtrace to the function which inputs and uses **ChanInDouble**, namely function **square**. Variables in scope with **square** now become available for inspection (at this stage they have not been initialised).

While still in function **square** move the cursor to the first line containing **ChanOutDouble** and set a breakpoint. Then press RESUME successively in order to run the program up to the breakpoint just set.

8.13.9 Inspecting by expression

In function **square** inspect the variable **sq** and check the computation by reinvoking INSPECT and specifying the expression **x * x**. Note how INSPECT can be used to perform arithmetic on any variable in scope. Expressions can also include numbers and other variables and constants in scope at the location point.

8.13.10 Modifying a variable

In breakpoint debugging any program variable (or even constant) may be modified. To modify a variable **x** press MODIFY and specify **x** at the 'Destination' prompt. The debugger now requests the new value by display the 'Source' prompt. Give any value and check the value has changed by inspecting **x** once again.

8.13.11 Backtracing to `main`

While still in **square**, press BACKTRACE to locate back to where the function was called. The debugger locates to **ProcPar** in function **main** where the five major processes are started in parallel. If the call to function **square** had been nested in other calls, successive BACKTRACE operations might have been necessary but would have eventually located to the call in the program main function.

8.13.12 Entering `#include` files

Press GOTO LINE and select line 20. This will locate you to the **#include <stdio.h>** line. By using the ENTER FILE key you may now enter the **#include** file (and any nested files within it); the EXIT FILE key will bring you out again into the enclosing file.

8.13.13 Quitting the debugger

Finally, to quit the debugger you should use the FINISH key. (You may also quit the debugger from the Monitor page using the 'Q' command).

8.14 Points to note when using the debugger

This section contains some extra information which may be of use when debugging parallel multiprocessor programs written in C.

8.14.1 Abusing hard links

Current generation transputers permit unsynchronised transfer of messages on external channels (links). This allows, for example, two 4-byte messages to be sent and for them to be received as a single 8-byte message on the receiving transputer. This is not consistent with the communication of messages between processes on the same processor where the transfer of messages is synchronised.

When breakpoint debugging, external communications are handled by the debugger's virtual link system; this is an internal transfer which is liable to function incorrectly if user code is relying on unsynchronised transfers.

Unsynchronised transfer of data should not be used where breakpointing is used to debug a program. It is bad practice anyway and will certainly cause the debugging virtual link system, on which breakpointing depends, to crash.

8.14.2 Examining the active network (the network is volatile)

When a process stops at a breakpoint you should remember that all of the other processes are still running (unless they hit a breakpoint, terminate etc.). This means that any of the Monitor page commands that display process queues (eg. `R`, `L`, `T` etc.) may change if you invoke them again (or use the '`U`' (Update) command to update the state information). When in symbolic mode the same is true for Channels which may appear empty when first inspected only to change to a waiting process when inspected again.

The only way to effectively *freeze* all processes is to flip to post-mortem mode by using the Monitor page '`Y`' (Enter Postmortem) command. You should remember that when you use this command that all processes that have hit a breakpoint will not appear in the runtime queues. If this is a problem, you should note the `Iptr` and `Wdesc` values of the processes and use the Monitor page '`O`' (Select Process) command to locate to them symbolically.

8.14.3 Selecting events from specific processors

The debugger provides no guarantee that debugging events such as breakpoints and debugging messages from processes running on different processors are presented in the same order in which they occur. Events on processors which are closer in terms of connectivity to the root transputer (where the debugger is running) are usually displayed before events on distant processors.

If it is important that you encounter a debugging event on a specific processor before events on other processors you can usually achieve this by changing to

the processor of interest (using the Monitor page 'P' command or left and right cursor keys) *before* resuming via the 'J' command.

8.14.4 Invalid pointers

The debugger checks instruction pointers (Iptrs) and workspace descriptors (Wdescs) for the correct code and data limits. Invalid pointers are flagged by an asterisk (*) on the screen.

Invalid pointers indicate a major problem with the program. They may also be caused by specifying an incorrect dump file.

8.14.5 INTERRUPT key

The debugger can be diverted from the running program to return to the Monitor page by the use of the INTERRUPT key. However, problems can arise if the running program is trying to simultaneously read keystrokes from the keyboard; the debugger is then unable to intercept the interrupt key. (Sometimes it is possible to force the interrupt to be recognised by repeating the key quickly.)

A similar problem arises when there are existing keystrokes buffered before the interrupt key; if the application program does not read these buffered keystrokes the debugger will never have a chance to see the interrupt key.

8.14.6 Program crashes

If in breakpoint mode the debugger detects that the program has crashed immediately after starting program execution (i.e. after invoking the 'J' (Jump into application) command), you should use the post-mortem breakpoint option ('X') to determine the cause. However, if no error flags are set on the network that is running the program then it is likely that the an error flag is set on a transputer that is not in use. This may occur on boards where the subsystem services are wired to propagate all error flags to the root transputer. In this instance you need to clear the network (see section 15.3.4 for more details).

8.14.7 Undetected program crashes

When operating in breakpoint mode and a program overwrites the debugging kernel or you have set a breakpoint in a high priority process on a processor without hardware breakpoint support, the debugger cannot fully recover and is unable to indicate that the program has crashed. This situation is indicated by the following message appearing at the top of the screen when the debugger attempts to display the Monitor Page:

Toolset Debugger : V2.00.00 Processor *n* "name" (T*m*)

In such instances you should use the host BREAK key in order to terminate the debugger and restart the debugger using the command line 'M' option to post-mortem debug the session.

8.14.8 Debugger hangs when starting program

If the debugger hangs immediately after you have supplied the command line arguments when starting execution of a program you have probably set a breakpoint in a configuration level High priority process on a processor without hardware breakpoint support.

8.14.9 Debugger hangs

If the debugger hangs when attempting to flip to post-mortem using the Monitor page 'Y' command or when trying to quit, you should terminate the debugger manually using the host BREAK key.

If you were trying to flip to post-mortem mode you should restart the debugger using the command line 'M' option to resume debugging in post-mortem mode.

8.14.10 Catching concurrent processes with breakpoints

Sometimes a concurrent process is executing in a program (often in a loop) and you would like to be able to control it better by use of breakpoints. If the process is communicating with other processes via channels and you have set breakpoints in the other processes, breakpoints can be set on a communication and the channel can be jumped down to the executing process when you hit the breakpoint.

However, if the process has entered a non-communicating loop or you are not sure where exactly it is in your program code you must use a different approach. In order to set a breakpoint, you should use the INTERRUPT key to return to the Monitor page and then, by using the 'R' (Run queues) command and/or the 'T' (Timer queues) command, list the `Iptrs` and `Wdescs` of the processes currently executing. (Often, this will include the debugging kernel processes but these are easily detected and ignored because they are marked by an asterisk.)

Use the 'G' (Goto process) command to select the `Iptr` and `Wdesc` of the process to locate symbolically to the process and set a breakpoint on that line. Then return to the Monitor page and resume the debugger using the 'J' command; when the process hits the breakpoint you may continue to debug it. If

there are no processes on either the runtime or timer queues and there are no external communications, it means that your program has either *deadlocked* or terminated.

8.14.11 Arrays as arguments

Because C requires a declaration of a parameter as *array of type* to be adjusted to *pointer to type* the debugger must treat all array parameters as pointers. This means that it cannot display the contents of an array of *arithmetic* type passed as a parameter automatically.

In order to display the contents of arithmetic arrays you should use array subranging. This is illustrated in the following example:

```
#include <misc.h>

void
foo (int p[4])

{
        /*  inspect p and p[0;3] here  */
        debug_stop ();
}


int
main (void)

{
        int p[4] = {0, 1, 2, 3};

        foo (p);
}
```

8.14.12 Backtracing with concurrent C processes

idebug supports backtracing from a parallel process to the parent process (where the parallel process was started via a C library call). However, for processes started asynchronously via **ProcRun**, **ProcRunHigh**, or **ProcRunLow**, **idebug** merely enables you to backtrace and does not allow operations such as inspection of variables after a backtrace. This is because the parent process which started the asynchronous processes may no longer exist, in which case inspection is meaningless.

8.14.13 Phantom breakpoints

Because of the mechanism used for breakpoints on those transputers without hardware breakpoint support (see Table 8.1) it is possible for code produced by INMOS compilers to contain code that fools the debugger into thinking it is a breakpoint (a phantom breakpoint). This may occur with `icc` and other TCOFF compatible INMOS compilers such as `oc`.

The following two fragments of code generate phantom breakpoints.

```
for (;;)    {
        ;
}

while (1)    {
        ;
}
```

If you encounter a phantom breakpoint and you wish to continue execution, you must set a breakpoint at the same address and then resume execution.

To do this use the GET ADDRESS key to obtain the start address of the empty loop when in symbolic mode, change to the Monitor page and use the Breakpoint Set option to set a breakpoint at the loop address.

8.14.14 Errors generated by the full library

Generally, the full C runtime library is able to detect when there is insufficient memory for it to function correctly; in such instances it displays an error message at startup.

In rare circumstances the library is able to detect that there is insufficient memory but it does not have enough memory to display the startup error message. In such instances, it sets the error flag and terminates execution.

If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger:

```
Error : Not compiled with debugging enabled "libc.lib"
```

then it is highly likely that insufficient memory is available for either the Static or Heap area.

8.14.15 Errors generated by the reduced library

Because the reduced C runtime library has no host to communicate with, if a runtime error occurs the reason for the error is not readily apparent.

If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger:

```
Error : Not compiled with debugging enabled "libcred.lib"
```

then it is highly likely that insufficient memory is available for either the Static or Heap area.

8.14.16 Shifting by large positive or negative values

Current transputers will temporarily 'lock' (for a time proportional to the shift value which is treated as unsigned) if you shift by large positive values or negative values. C performs no runtime checks for invalid shift values and does not protect you against their consequences. (Certain transputer languages such as occam do perform these checks).

If the debugger when used in post-mortem mode locates to a source line containing a shift operator and the error flag has not been set then it is likely that you have shifted by an invalid value.

8.14.17 Compiler optimisations

`icc` performs some code optimisations. If an external variable is optimised out from a module because it is never used the debugger is informed of this and is able to relay this information to the user.

However, for some optimisations the debugger is not informed and consequently it may provide misleading information. The following code illustrates this:

```
int
main (void)

{
        int     a = 0;
        int     b = 0;

        while (1)   {   /*  or  for (;;)  */
                ;
        }

        /*  following optimised out by compiler  */
        a = 42;
        b = a + 1;
        a = b * b
        ...
}
```

In these cases the debugger may show the discrepancy in either of the following ways:

1 If a function follows the optimised code the debugger associates the address of the optimised lines with the address of the start of the function.

2 If no function follows the optimised code the debugger indicates that it is unable to find the address for any of the optimised lines.

8.14.18 Determining connectivity and memory sizes

In order to establish the connectivity and memory map range for each processor in a program you should use the debugger command line dummy debug session 'D' option.

You should remember for non-configured programs that the memory map requirements may be larger than those indicated because of initialisation processes which are overlayed.

9 Mixed language programming

This chapter describes the mechanisms supplied with the toolset for mixing code modules written in different high level languages. It describes both the generalised system for mixing code at configuration level and the special facilities that support the incorporation of occam code into C programs.

9.1 Introduction

For many applications it is appropriate to write the software using more than one programming language. For example, a particular algorithm may be better expressed in a specific language or applications software may already exist in particular languages. In either case a well defined mechanism for mixing languages within a single system is desirable.

The communicating process programming model provides a clean and simple basis for mixing languages. The model consists of independent processes, communicating via channels, which can be distributed in any way to a network of transputers using a configuration description. Programs can be written in different languages, compiled and linked using a common set of tools, and the linked units placed anywhere on a network of transputers.

Programs written using any of the INMOS compilers and toolsets which generate code in compatible TCOFF format can be freely mixed in the same configuration as linked modules.

A special mechanism supports the importation of occam procedures and functions into C programs, based on a C compiler pragma. A pragma is also provided for translating occam names into valid C names.

A set of interfaces is also provided for incorporating code written using earlier INMOS 3L toolsets. The interfaces use a series of occam harnesses for different types of C program and are described in appendix F.

9.2 Mixing code at configuration level

The mixing of code written in different languages can be achieved at the configuration level, using linked units generated using any of the INMOS TCOFF compilers. The TCOFF family of compilers generates object code in a special format which is interchangeable at configuration level.

9.2.1 C and occam

Linked object files which are to be configured can contain entirely C code, entirely occam code, or mixtures of the two. Remember when linking any C code to also link in the appropriate linker startup file (`startup.lnk` or `started.lnk` depending whether the program uses the full or the reduced library), and when linking any occam code remember to link in the compiler libraries. Linker indirect files which specify the correct occam compiler library for different transputer targets are supplied with the TCOFF version of the occam 2 toolset (IMS D*X*205).

The configuration description allows complete flexibility in the placement of software modules onto the hardware network. It can be used, for example, to place processes written in different languages on the same processor as easily as on a network of processors interconnected by transputer links. Each code module must be a fully linked unit in which all external references are already resolved and must have been created in TCOFF format.

For further information about the configuration system and language, including examples of simple configuration descriptions, see chapter 6.

9.3 Calling occam processes

Special facilities are provided in the toolset to allow occam procedures and functions to be imported into C programs as C functions. The mechanism uses the `icc` pragma `IMS_nolink` to prevent the addition of a static link parameter when the call to the occam function is compiled.

9.3.1 Pragma `IMS_nolink`

`IMS_nolink` disables the passing of the global static base (gsb) parameter when the occam code is called. The gsb locates the static area for C functions but would disrupt the normal occam calling sequence.

For example, consider the occam function `ocfunc` which performs some unspecified calculation and returns a single integer value:

```
INT FUNCTION ocfunc(VAL INT arg1, arg2)
  INT ret:
  VALOF
    SEQ
     -- calculate ret
  RESULT ret
:
```

To call **ocfunc** from a C program it must first be declared as an **extern** function and then specified as having no static base parameter:

```
extern int ocfunc(int arg1, int arg2);
/* declare function as extern */

#pragma IMS_nolink(ocfunc)
/* direct function to be compiled
   with no static base parameter */

void call_oc(void)
{
  int arg1, arg2, ret;
  /* set up arguments */

  ret = ocfunc(arg1, arg2);
  /* call function */
}
```

When linking the C program the file containing the occam function must be linked with the program in the same way as any other compiled object file. Remember to link in the occam compiler libraries (using the appropriate linker indirect file for the transputer type, supplied with the occam toolset) and any other libraries that the occam program uses.

An alternative to using the nolink pragma is to compile the occam code with a dummy first parameter of type **INT**. The dummy parameter is not used by occam and simply ensures compliance with the C calling requirements.

9.3.2 Translating occam names

The compiler pragma option **IMS_translate** is provided to allow occam names, such as those containing invalid C characters, to be replaced by an acceptable C alias. For example, it is common in occam to use the full stop character to create multi-part names. Use of the full stop is prohibited in C.

The pragma allows occam identifiers, where it is impossible or undesirable to change them, to be referenced in the program by valid C names. The syntax is as follows:

```
#pragma IMS_translate(Cname, "occamname")
```

For example:

```
#pragma IMS_translate(occam_func "occam.func")
```

All references to `occam_func` in the source code will be translated into `occam.func` in the object file.

9.3.3 Rules for importing occam code

1 Only occam procedures and occam functions returning a single value, may be called.

2 The occam process to be called must be at the outer level of a separately compiled unit.

3 All interaction with the calling program must be via channels.

4 No process which requires direct communication with the host file server may be called.

5 Formal occam parameters, return values from occam functions, must be mapped onto actual C parameters of the correct type. The calling conventions are described in section 9.4.

6 The occam process must not use vector space, or call any other occam process which uses vector space. If arrays are used they should be explicitly placed within the workspace. occam libraries supplied with the occam 2 toolset which use vector space and therefore cannot be called from C are: **`hostio.lib`**, **`streamio.lib`**, **`process.lib`**, **`msdos.lib`**, and **`streamco.lib`**.

7 There must be enough workspace on the stack of the calling C program. This must be ensured by the programmer.

8 Non-VAL occam parameters should be passed as pointers from C.

9 Where the formal parameter to an occam procedure or function is an array (VAL or Non-VAL) the calling C program should always pass a *pointer* to the array. For an occam array parameter with unspecified array bounds, the actual sizes of the bounds should be passed immediately following the array parameter; for multidimensional arrays the bounds should be passed in the same order as they appear in the declaration.

For example, to call the following occam procedure (which uses a bounded array):

```
PROC ocproc([8]INT array)
```

the following code should be used:

```
extern void ocproc(int array[8]);
#pragma IMS_nolink(ocproc)
int array[8];
ocproc(array);
```

To call the following occam procedure (unbounded array):

```
PROC ocproc([]INT array)
```

use the C code:

```
extern void ocproc(int array[], int arraysize);
#pragma IMS_nolink(ocproc)
int array[8];
ocproc(array, 8);
```

9.4 Parameter passing

The following tables describe the calling conventions that must be followed when passing parameters from C programs to imported occam processes. They list the C equivalents on 32 and 16 bit transputers for all occam types. Where there is no true equivalent the action to take is given.

Formal occam parameter	Actual C parameter	
	(32 bit)	(16 bit)
VAL BOOL	int (value must be 0 or 1)	int (value must be 0 or 1)
VAL BYTE	char unsigned char	char unsigned char
VAL INT16	short int	short int int
VAL INT32	int long int	long int *
VAL INT64	No direct equivalent†	No direct equivalent†
VAL INT	int	int
VAL REAL32	float	float *
VAL REAL64	double *	double *
VAL array	array (see above)	array (see above)

† There is no direct type equivalent in C. Either recode the occam program or pass the parameter in another form.

Formal occam parameter	Actual C parameter	
	(32 bit)	(16 bit)

BOOL	`char *`
	`unsigned char *`
	value pointed to must be 0 or 1)
		`char *`
		`unsigned char *`
		value pointed to must be 0 or 1)

BYTE	`char *` `unsigned char *`	`char *` `unsigned char *`
INT16	`short int *`	`short int *` `int *`
INT32	`int *` `long int *`	`long int *`
INT64	No direct equivalent†	No direct equivalent†
INT	`int *`	`int *`
REAL32	`float *`	`float *`
REAL64	`double *`	`double *`
CHAN	`Channel *` (see Note 1)	`Channel *` (see Note 1)
PORT	No direct equivalent†	No direct equivalent†
TIMER	Pass nothing (see Note 2)	Pass nothing (see Note 2)
array	`array` (see above)	`array` (see above)

† There is no direct type equivalent in C. Either recode the occam program or pass the parameter in another form.

Note 1: `Channel` is an INMOS specific type declared in the header file `channel.h`.

Note 2: An occam `TIMER` parameter should have no associated C actual parameter passed. For example, to call the occam procedure:

```
PROC ocproc(VAL INT p1, TIMER t, VAL INT p2)
```

use the following C call:

```
extern void ocproc(int p1, int p2);
#pragma IMS_nolink(ocproc)
ocproc(p1, p2);
```

9.4.1 Return values

The following table outlines the conventions that must be followed when receiving occam function return values in C.

occam function type	C function type	
	(32 bit)	(16 bit)
BOOL	`int`	`int`
BYTE	`char` `unsigned char`	`char` `unsigned char`
INT16	`short int`	`short int` `int`
INT32	`int` `long int`	`long int`
INT64	No direct equivalent†	No direct equivalent†
INT	`int`	`int`
REAL32	`float`	`float`
REAL64	`double`	`double`
† There is no direct type equivalent in C. Either recode the occam program or return the value in another form.		

9.4.2 Example of passing parameters

The following example shows an occam function with a variety of formal parameters, along with the C code which can call it. The calling code for 32 bit and 16 bit transputers is given separately.

The occam function to be called is as follows:

```
INT32 FUNCTION ocfunc1(VAL INT32 parm1) IS parm1:

PROC ocproc1(VAL BYTE vb,
             VAL INT16 vi16,
             VAL INT32 vi32,
             VAL INT vi,
             VAL REAL32 vr32,
             VAL REAL64 vr64,
             VAL BOOL vbo,
             VAL []INT varr1,
             VAL [8]INT varr2,
             BYTE b,
             INT16 i16,
             INT32 i32,
             INT i,
             REAL32 r32,
             REAL64 r64,
             BOOL bo,
             []INT arr1,
             [8]INT arr2)
  SEQ
    b := vb
    i16 := vi16
    i32 := vi32
    i := vi
    r32 := vr32
    r64 := vr64
    bo := vbo
    arr1 := varr1
    arr2 := varr2
:
```

The C code to call the above occam function on a 32 bit transputer is as follows:

```
#define ARRAY_SIZE_1 4
#define ARRAY_SIZE_2 8

extern long int ocfunc1(long int parm1);

extern void ocproc1(char vb, short int vi16,
                    long int vi32, int vi,
                    float vr32, double *vr64,
                    int vbo,
                    int varr1[], int varr1_size,
                    int varr2[ARRAY_SIZE_2],
                    char *b, short int *i16,
                    long int *i32, int *i,
                    float *r32, double *r64,
                    char *bo,
                    int arr1[], int arr1_size,
                    int arr2[ARRAY_SIZE_2]);

#pragma IMS_nolink(ocfunc1)
#pragma IMS_nolink(ocproc1)

long int li, result;
char vb, b;
short int vi16, i16;
long int vi32, i32;
int vi, i;
float vr32, r32;
double vr64, r64;
int vbo;
char bo;
int varr1[ARRAY_SIZE_1], arr1[ARRAY_SIZE_1];
int varr2[ARRAY_SIZE_2], arr2[ARRAY_SIZE_2];

result = ocfunc1(li);

ocproc1(vb, vi16, vi32, vi, vr32, &vr64,
        vbo, varr1, ARRAY_SIZE_1, varr2,
        &b, &i16, &i32, &i, &r32, &r64,
        &bo, arr1, ARRAY_SIZE_1, arr2);
```

The C code to call the above occam function on a 32 bit transputer is as follows:

```
#define ARRAY_SIZE_1 4
#define ARRAY_SIZE_2 8

extern long int ocfunc1(long int *parm1);

extern void ocproc1(char vb, short int vi16,
                    long int *vi32, int vi,
                    float *vr32, double *vr64,
                    int vbo,
                    int varr1[], int varr1_size,
                    int varr2[ARRAY_SIZE_2],
                    char *b, short int *i16,
                    long int *i32, int *i,
                    float *r32, double *r64,
                    char *bo,
                    int arr1[], int arr1_size,
                    int arr2[ARRAY_SIZE_2]);

#pragma IMS_nolink(ocfunc1)
#pragma IMS_nolink(ocproc1)

long int li, result;
char vb, b;
short int vi16, i16;
long int vi32, i32;
int vi, i;
float vr32, r32;
double vr64, r64;
int vbo;
char bo;
int varr1[ARRAY_SIZE_1], arr1[ARRAY_SIZE_1];
int varr2[ARRAY_SIZE_2], arr2[ARRAY_SIZE_2];

res = ocfunc1(&li);

ocproc1(vb, vi16, &vi32, vi, &vr32, &vr64,
        vbo, varr1, ARRAY_SIZE_1, varr2,
        &b, &i16, &i32, &i, &r32, &r64,
        &bo, arr1, ARRAY_SIZE_1, arr2);
```

9.5 Mixing code using the occam 2 toolset

If you also have the occam 2 toolset installed, code written in different languages can be mixed with current TCOFF-compatible code using a special interface code system. The facility extends to code written using the earlier INMOS 3L compilers and toolsets.

Any non-occam code can be wrapped in an occam envelope and treated as an equivalent occam process, providing that certain interfacing rules are applied. The system is similar to that described in the '*occam 2 toolset user manual*', but with some modifications to the interfaces. Details of the interfaces supported for this form of mixed language programming are given in appendix F.

9.5.1 Calling C from occam

The library `callc.lib` can be used to call C programs from occam. The library is provided with the occam toolset (TCOFF based version) and information about its use can be found in the '*occam 2 toolset user manual*'.

10 Using the EPROM tools

10.1 Introduction

INMOS EPROM software is designed so that programs can be developed and tested using the INMOS toolset, and once they are working, can be placed in ROM with only minor change.

Under development, software is booted onto a network from a link connecting the network to the host computer. Then the software is prepared for a ROM, which is attached to the root transputer in the network.

Figure 10.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

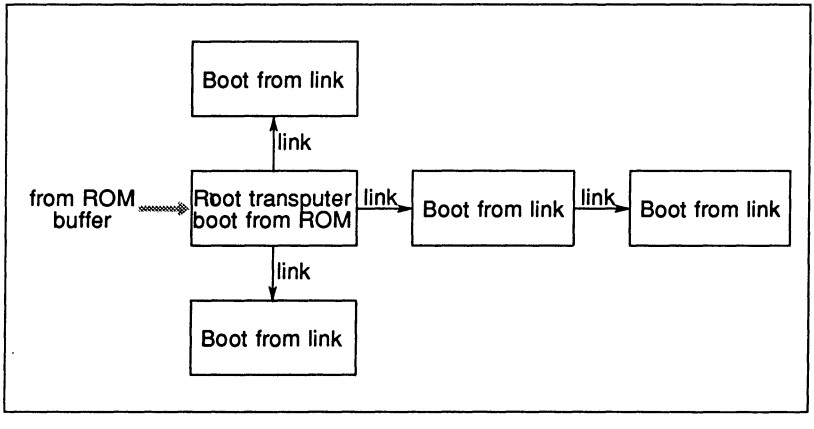


Figure 10.1 Loading a network from ROM

To prepare software to be booted from ROM, rather than to be booted from link, the following two steps must be taken:

1 Give different options to the configurer and collector tools so that they produce ROM-bootable code.

2 Run the **ieprom** tool to produce a file or set of files suitable for blowing into EPROM.

Figure 10.2 illustrates the stages of preparing ROM-bootable software. One or

more linked units will be referenced from the configuration file, depending on whether it is a single or multi-process program.

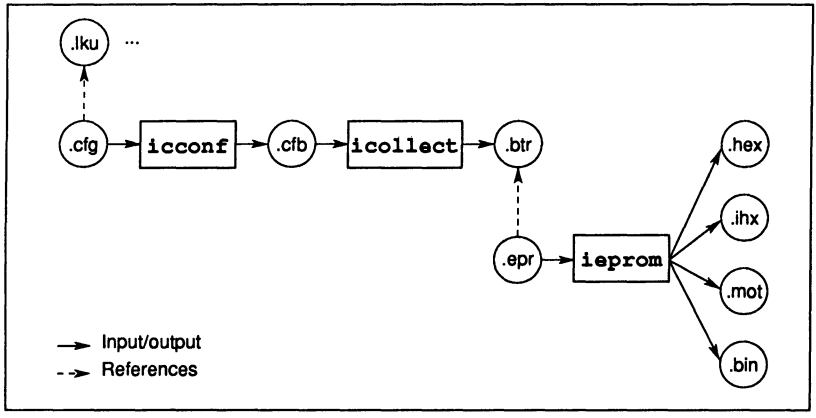


Figure 10.2 Preparation stages for ROM-bootable software

10.2 Processing configurations

The processing configuration used will depend on the number of software processes, the number of transputers available to run the code and whether the code is to run from ROM or RAM. The following sections outline the possible configurations.

10.2.1 Single process, single processor, run from ROM

The application process is prepared as a single configuration process. (See section 10.4). The application process is then run in the processor, directly from ROM, using the RAM as the data area for static variables, workspace and heap.

10.2.2 Multiple process, single processor, run from ROM

The application is prepared as a collection of processes, connected together as described in a configuration file. It is then run on a single processor, with the code in ROM, and the RAM is used as the data area.

10.2.3 Single process, single processor, run from RAM

The application is prepared as a single configuration process (See section 10.4). When booted from ROM, the processor loads the code into RAM, and executes it there; the data area is also in RAM.

10.2.4 Multiple process, single processor, run from RAM

The application is prepared as a collection of processes, connected together as described in a configuration file. When booted from ROM, the processor loads the code for all the processes into RAM, and sets them all running, with their data areas also in RAM.

10.2.5 Multiple process, multiple processor, run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads its own code into RAM, and loads the rest of the network via its links. Each processor then sets off its own processes, and the application runs. (This configuration is shown in figure 10.1).

10.2.6 Multiple process, multiple processor, root run from ROM, rest of network run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads the rest of the network via its links, and then continues to run its own code from ROM.

10.3 The eprom tool: `ieprom`

The eprom tool **`ieprom`** takes the output of the collector, and produces a file or set of files suitable for blowing into an EPROM. The following output formats are supported:

- Binary
- Hex
- Intel hex format

- Intel extended hex format

- Motorola S-record format

ieprom supports the production of code files in *block mode*, which allows the code to be placed in a set of different files. This is useful to program EPROMS organised as separate byte-wide devices, or where the EPROM programming device does not have enough memory to hold the entire image.

ieprom also supports the inclusion in the EPROM image of a *memory configuration*. Some 32-bit transputers have a configurable memory interface which can be initialised from a fixed area in the ROM, when the transputer is reset. A particular memory configuration can be specified to **ieprom** in a text file. These files are known as memory configuration files and normally have the file extension **.mem**. The format of these files, and the facility to edit them using an interactive tool called **iemit** is described in chapter 17.

The **ieprom** tool is driven by a control file which normally has the file extension **.epr**. A detailed description of **ieprom** and its control file is given in chapter 18.

10.4 Using the configurer and collector to produce ROM-bootable code

To produce code suitable for running in ROM or RAM, the configurer and collector tools must be specified with the appropriate command line options. The following options are used for both tools:

- The **ro** option specifies that the code is to run in ROM.
- The **ra** option specifies that the code is to run in RAM.
- The **rs** option specifies the ROM size.

In addition the **p** option must be specified for the configurer, in order to specify the root processor name.

The collector will add the appropriate ROM bootstrap to the application code and the output file will be given the extension **.btr**.

When preparing code to run in ROM or RAM, the configuration phase must be used, in order to specify the size of stack and heap to be used. This applies even when the application consists of a single process running on a single processor.

10.5 Summary of EPROM steps for different processing configurations

	Compile and link	Configure	Collect	EPROM
Single process, single processor, run from ROM.	Compile and link program as a single unit.	Configure with the **-ro**, **-rs** and **-p** options.	Collect with the **-ro** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple process, single processor, run from ROM.	Compile and link a set of units, one per process.	Configure with the **-ro**, **-rs** and **-p** options.	Collect with the **-ro** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single process, single processor, run from RAM.	Compile and link program as a single unit.	Configure with the **-ra**, **-rs** and **-p** options.	Collect with the **-ra** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple process, single processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the **-ra**, **-rs** and **-p** options.	Collect with the **-ra** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple process, multiple processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the **-ra**, **-rs** and **-p** options.	Collect with the **-ra** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple process, multiple processor root runs from ROM, rest of network runs from RAM.	Compile and link a set of units, one per process.	Configure with the **-ro**, **-rs** and **-p** options.	Collect with the **-ro** and **-rs** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

Tools

11 icc – ANSI C compiler

This chapter describes in detail the ANSI C compiler `icc`. It describes the command line syntax, compiler options, and preprocessor directives, explains what is meant by transputer classes and how to use them, and describes other features of the compiler such as support for transputer code. The chapter ends with a list of error messages.

11.1 Introduction

The ANSI C compiler is a full ANSI C compiler with support for concurrent programming. It also supports some additional extensions to the C language including compiler directives, pragmas and low level programming.

The ANSI standard for the C language extends the language through the definition of runtime library support, new types, function prototyping, and many other ways. For a summary of the differences between ANSI C and the original definition of the language see chapter 3 '*New features in ANSI C*' in the accompanying reference manual. The ANSI C compiler includes support for parallel programming through a set of library functions with associated types and structures, a mechanism for incorporating transputer code sequences, and a group of compiler pragmas for enabling compiler options in sections of code and for conveying directives to the linker. The transputer code mechanism supports the full set of transputer instructions and operations and also supports labels.

Parallel processing is achieved through a library of process, channel, and semaphore functions and their related types and data structures. Calls to the functions are compiled by `icc` into highly efficient parallel code for the transputer.

`icc` is itself written in ANSI C and normally runs on the transputer board. A version of the compiler running on the host system is supplied with Sun and VAX based toolsets.

`icc` generates code for a particular transputer, transputer type, or class, and a target should be specified for all compilations. The default is to produce code for the IMS T414.

11.2 Running the compiler

To invoke the compiler use the following command line:

▶ **icc** *filename* {*options*}

where: *filename* is the C program source file. If no extension is given **.c** is assumed. Only one filename may be given on the command line.

options is a list of the options given in the following tables.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Note: **icc** must be invoked in a writeable directory, that is, one in which you (or any alias you use to invoke the compiler) have *write* access.

Examples of use:

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icollect hello.lku -t
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icollect hello.lku /t
iserver /sb hello.btl /se
```

Option	Description
TA	Compile for transputers of class A (T400, T414, T425, T800, T801, T805).
TB	Compile for transputers of class B (T400, T414, T425).
T212	Compile for T212 transputer.
T2	Compile for T212, T222, or M212 transputers.
T222	Compile for T222 transputer.
T225	Compile for T225 transputer.
T3	Compile for T225 transputer.
T400	Compile for T400 transputer.
T414	Compile for T414 transputer. This is the default processor type and may be omitted if the target processor is a T414.
T4	Compile for T414 transputer.
T425	Compile for T425 transputer.
T5	Compile for T425 or T400 transputers.
T800	Compile for T800 transputer.
T8	Compile for T800 transputer.
T801	Compile for T801 transputer.
T805	Compile for T805 transputer.
T9	Compile for T801 or T805 transputers.
C	Performs a syntax check only. Generates no object code.
D *symbol*	Defines a symbol. Same as **#define** *symbol* **1** at the start of the source file.
D *symbol=value*	Defines a symbol and assigns a value. Same as **#define** *symbol value* at the start of the source file.
EC	Disables checks for invalid type casts. ANSI compliance check.
EP	Disables checks for invalid text after **#else** or **#endif**. ANSI compliance check.
EZ	Disables checks for zero-sized arrays. ANSI compliance check.
FH	Checks that all **extern** function definitions are preceded by a declaration and reports all unused forward **static** declarations. Software QA check.
FM	Generates warning messages on **#defined** but unused macros.
FV	Generates warning messages on declared but unused variables or functions (default).

Option	Description
G	Generates comprehensive debugging data. The default is to produce minimal debugging data. Debugging data is required for the correct operation of **idebug**.
I	Displays detailed progress information at the terminal as the compiler runs.
J *dir*	Adds *dir* to the list of directories to be searched for source files incorporated with the **#include** directive in extended search paths. See section 11.3.1 for details.
L	Loads the tool onto the transputer board and terminates.
KS	Enables stack checking.
O *outputfile*	Specifies an output file. If no filename is given the compiler derives the output filename from the input filename stem and adds the **.tco** extension.
PP	Lists the preprocessed source file to **stdout**.
S	Compiles the source file to assembly language and writes it to a file. Assembly is suppressed and no object code is produced. The file is named after the input file and given the **.s** extension.
U *symbol*	Disables a symbol definition. Equivalent to **#undef** *symbol* at the start of the source file.
WA	Suppresses messages warning of '=' in conditional expressions.
WD	Suppresses messages warning of deprecated function declarations.
WF	Suppresses messages warning of implicit declarations of **extern int()**.
WN	Suppresses messages warning of implicit narrowing or lower precision.
WT	Suppresses messages warning of the possibility of less efficient code when compiled for a transputer class.
WV	Suppresses messages warning of non-declaration of **void** functions.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

11.2.1 Transputer targets

The compiler generates code for a specific transputer type. This means that a processor type should be specified for all transputer targets except the default that is built into the compiler. The default processor type which is used if no target is specified is T414.

Processor types supported are the IMS T212, M212, T222, T225, T400, T414, T425, T800, T801 and T805 transputers. For the purpose of generating common code for several transputer types these are also grouped into transputer classes.

Transputer classes group transputers according to word size, the position of the start of usable memory, and instruction set compatibility. They can be used to generate code for combinations of transputers.

Details of transputer types and classes can be found in section 5.3.

11.2.2 Error modes

All code in mixed language transputer programs must be compiled and linked in the same or a compatible error mode. `icc` always generates code in UNIVERSAL error mode, which is compatible with HALT and STOP error modes created by other INMOS compiler toolsets.

The error mode for a mixed language program can be consolidated into a single mode for the entire program by specifying the appropriate linker option. If no mode is specified the linker generates the program in HALT mode.

11.2.3 Default command line options

Commonly used command line parameters can be defined in the host environment variable `ICCARG`. Parameters specified in this way are automatically added to the command line when the compiler is invoked.

Command line parameters must be specified in `ICCARG` using the syntax required by the `icc` command line.

11.2.4 File extension defaults

The `.c` extension is assumed on input source files and does not need to be specified. If no output file is specified the compiled object file is named after the input file and given a `.tco` extension. A `.tco` extension is also added if a file is specified without an extension.

11.2.5 Search paths

The normal search paths are used for locating files specified on the command line. The search rules are described in section A.3.

Search paths for files imported with the **#include** compiler directive differ slightly from those for files specified on the command line and can be extended by the use of special syntax and a command line option. Details of this facility can be found in section 11.3.1.

11.3 Compiler directives

11.3.1 #include

Syntax: **#include** *filename*

The **#include** directive instructs the preprocessor to copy the contents of the named file into the current file. The filename must be enclosed within angle brackets (*<filename>*) or double quotes (*"filename"*). The two forms of syntax generate different search strategies.

If angle brackets are used *only* those directories specified by **ISEARCH** are searched. No other directories (including the current directory) are searched. This system is mainly used to include the standard library header files.

If double quotes are used to enclose the filename the standard toolset search is used, but incorporating a method for extending the search list. First the current directory is searched. If the file is not found the search continues with the list of directories specified after the compiler '**J**' option. If the file is still not found, or if no list is given, directories specified by **ISEARCH** are searched in the normal way.

Relative directory names

Relative directory names are treated as relative to the directory containing the current source file.

Backslash character in filenames

In included filenames the backslash is not treated as an introducer to an escape sequence unless it is followed by another backslash ('\\').

11.3.2 #define

Syntax: **#define name** [(*arg1,...,argn*)] [*value*]

The define directive allows simple macro substitution to be performed. In its simplest mode of operation *name* and *value* represent a series of ASCII characters causing the preprocessor to substitute all occurrences of *name* by *value* (which may be NULL). Arguments may also appear after the name, and when this happens the preprocessor will still replace all occurrences of *name* and its following arguments by *value*, but in this case the value string will have been defined in terms of the expected arguments, and will therefore exhibit a dependence on the original text.

```
#define YES 1      /* replace all occurrences
                      of YES by 1 */

#define max(a,b) (a > b ? a : b)
  /* max(2,4) will be replaced by
     (2 > 4 ? 2 : 4) */
```

11.3.3 #undef

Syntax: **#undef** *identifier*

This directive causes the current definition of *identifier* (as defined using the **#define** directive) to be deleted.

11.3.4 #if

Syntax: **#if** *constant_expression*

This directive, along with the **#else** and **#endif** directives, is used in a similar way to the if ... else construct of many high level programming languages. When it is encountered, the preprocessor evaluates the following constant expression and if it is zero it deletes all text up to the following **#else** or **#endif** directive. If, however, the expression evaluates to non-zero, then only the text between the **#else** and **#endif** directives (if any) is removed. This mechanism would typically be used to allow conditional compilation.

As an extension to this directive, the preprocessor also allows 'if defined' typed expressions. In this case 'defined' is used as a unary operator which returns true if its operand represents an identifier that is currently defined within the preprocessors symbol table, and false if it is not. By combining this operator with the logical operators it is possible to build complex expressions of the form:

```
#if defined foo & ! defined dummy

    /* if foo is defined & dummy is not */
```

11.3.5 #ifdef

Syntax: **#ifdef** *identifier*

This directive works in a similar way to the **#if** directive, but instead of basing its decision on the result of an expression it uses the existence or non-existence of the identifier within the preprocessors' symbol table as the criterion. If the identifier has not previously appeared in a **#define** directive then all text up to the following **#else** or **#endif** directive is deleted; otherwise all text between the **#else** and **#endif** directives is removed.

11.3.6 #ifndef

Syntax: **#ifndef** *identifier*

This directive is similar to **#ifdef**, except that the text is passed if *identifier* is not currently defined.

11.3.7 #else

Syntax: **#else**

This directive can be used with the **#if**, **#ifdef**, and **#ifndef** directives to mark the beginning of text which will be ignored whenever the expression following the **#if** evaluates to a non-zero value.

11.3.8 #elif

Syntax: **#elif**

This directive can be used in place of the sequence **#else #if**.

11.3.9 #endif

Syntax: **#endif**

This directive must be used with the **#if**, **#ifdef**, and **#ifndef** directives to mark the end of the text which may be affected by the **#if** ... **#else** ... **#endif**

construct.

11.3.10 #line

Syntax: **#line** *linenumber* [*filename*]

This directive instructs the compiler that it is currently processing line number *linenumber* in the file *filename*. If no file name is specified, the original name is retained.

11.3.11 #pragma

Syntax: **#pragma** *pragma* (*params*)

This directive activates and deactivates various compiler options in sections of C code. It may be used to set (or override) options specified on the command line. Most pragmas also take parameters or numerical arguments.

The following two tables list the main compiler pragmas and the parameters to **IMS_on** and **IMS_off**.

Option	Description
IMS_on (*params*)	Enables specific compiler checks. Takes a list of parameters which specify the checks to be enabled.
IMS_off (*params*)	Disables compiler checks. Takes a list of parameters which specify the checks to be disabled.
IMS_nolink (*functionname*)	Compiles the function *functionname* without a static link parameter. The function must already have been declared but must not have been defined or called. This pragma is used for importing code written using languages such as occam which do not use static data, and for exporting C functions to the same languages.
IMS_linkage (["*name*"])	Adds ordered reference tags to specific regions of code. The tags are directives to the linker which force a specific segment ordering. For further details about link time ordering see section 20.3.1.

Option	Description
`IMS_modpatchsize(`*n*`)`	Specifies the number of bytes reserved by the compiler for a linker module number patch. *n* has default values of 3 for 32-bit targets and 2 for 16-bit targets.
`IMS_codepatchsize(`*n*`)`	Specifies the number of bytes *n* reserved by the compiler for a linker code patch. *n* has a default value of 6 for 32-bit targets and 4 for 16-bit targets.
`IMS_translate (`*name*`, "`*newname*`")`	Directs the compiler to replace all references to *name* (for example an external routine) by *"newname"*. *"newname"* is a C string which can contain alphanumeric characters and the underscore ('_'), percent ('%'), and full stop ('.') characters.

Parameters to `IMS_on` and `IMS_off`:

Parameter	S/f	Description
`channel_pointers`	`cp`	Treats a variable of type `Channel` in the scope of the definition `typedef void *_IMS_Channel` as a channel type for the debugger. Default is off. This pragma is enabled in the header file `channel.h`. If `channel.h` is included in the program this pragma will remain active until specifically disabled.
`inline_ops`	`il`	Compiles certain operations on `long` operands (signed or unsigned) on 16-bit targets as in line operations rather than as calls to the compiler library. Operations affected are: ~ (bitwise complement), +, –, & (bitwise AND), | (bitwise OR), ^ (bitwise exclusive OR), <<, >>, <, <=, ==, !=, >=, and >. Default is on.

Parameter	S/f	Description
inline_string_ops	**is**	Compiles the library functions **memcpy** and **strcpy** as in-line transputer code under certain conditions. For further details see section 11.4. Default is off. **inline_string_ops** is enabled in the standard header file, **string.h**, where **memcpy** and **strcpy** are declared.
printf_checking	**pc**	Checks that arguments passed to a function conform to the format used by **printf**. Default is off. This pragma is normally used to check formal arguments which are to be passed directly as format strings to **printf**. For each function within the scope of the pragma the last formal parameter is read as a format string and subsequent variable arguments are checked for correct type, according to the formatting rules of **printf**. This pragma is enabled in **stdio.h** for the declaration of **printf** and related functions, and subsequently disabled.
scanf_checking	**sc**	Checks that arguments passed to a function conform to the format used accepted by **scanf**. Default is off. Otherwise this pragma has the same effect **printf_checking**. This pragma is enabled in **stdio.h** for the declaration of **scanf** and related functions, and subsequently disabled.
stack_checking	**sc**	Checks for stack overflow at the start of each function. Default is off.
warn_bad_target	**wt**	Warns of inferior code generated for a transputer class rather than for a specific transputer target. Default is on.
warn_deprecated	**wd**	Warns of parameterless function declarations. Default is on.
warn_implicit	**wi**	Warns of undeclared functions. Default is on.

Pragma IMS_nolink

The pragma **IMS_nolink** enables C routines to call or be called from occam and other languages.

Syntax: **#pragma IMS_nolink (***fname***)**

The following code uses the pragma to allow an occam routine OCCAMREALOP to be called in a C program:

```
extern float OCCAMREALOP(const float x,
                         const int op,
                         const float y);
#pragma IMS_nolink (OCCAMREALOP)
:
:
float x, y, z;
z = OCCAMREALOP(x, op_add, y);
```

The following code allows the C function **max** to be called from occam:

```
extern int max(const int x, const int y);
#pragma IMS_nolink (max)
extern int max(const int x, const int y)
{ return x > y ? x : y; }
```

11.3.12 #error

Syntax: **#error** *text*

This directive causes an explicit error. If there is no pragma in force, the compiler terminates immediately and the text following the directive is displayed on the screen. If a pragma is already in force, the text is displayed but the compilation is not aborted. This is useful for determining which pieces of code are being bypassed by a construct of the form **#if ... #else ... #endif**.

11.4 Optimised functions

Optimised versions of **memcpy** and **strcpy** are provided in the form of the library functions **_memcpy** and **_strcpy**. These functions are compiled directly in-line as transputer code under certain conditions, thereby optimising their performance.

```
void *_memcpy(void *dest, const void *source,
              size_t n);

char *_strcpy(char *dest, const char *source);
```

_memcpy is compiled directly as a transputer block move operation when **n** is a positive integer constant and either no result is required or **dest** is a simple local pointer. The value of **n** must be positive because the result of the block move operation is undefined with a string length of zero.

_strcpy is compiled directly as a transputer block move operation when **source** is a string literal and either no result is required or **dest** is a simple local pointer.

If the pragma **inline_string_ops** is enabled, calls to the ANSI standard functions **memcpy** and **strcpy** are treated as calls to **_memcpy** and **_strcpy**, and will consequently be compiled in line if the required conditions are met.

Note: **inline_string_ops** is enabled in the standard header file **string.h**, which also declares **memcpy** and **strcpy**. If this header file is included in the source then calls to **memcpy** and **strcpy** will automatically be treated as calls to the respective in line functions and compiled as transputer code.

11.5 Compiler predefinitions

Certain constants which identify global information, and some function names, are automatically recognised by the compiler. Generally, these items can be referenced directly in C programs do not need to be declared.

Note: Predefined functions **_lsb** and **_params** (see section 11.5.3) should be declared to avoid spurious warning messages being generated by the compiler.

11.5.1 Constants

All predefined constants defined by the ANSI standard are present.

The following INMOS constants are also defined:

__CC_NORCROFT	–	Norcroft C compiler
_ICC	–	ANSI C compiler
_PTYPE	–	Processor type
_ERRORMODE	–	Execution error mode

Details of the constants and the values they can take can be found in chapter 4 of the accompanying Reference Manual.

11.5.2 Functions

The optimised library functions `_memcpy` and `_strcpy` are predefined.

11.5.3 Other predefines

Two further names `_lsb` and `_params` are predefined by the compiler. They can be used in expressions in the same way as C variables. Both represent addresses which may be manipulated in low level programming.

```
volatile const void *_lsb

volatile const void *_params
```

`_lsb` is a pointer to the base of the compiled file's data area.

`_params` is a pointer to the base of the the current function's parameter block. It can be used to obtain low level information about a function's runtime code.

The following example illustrates how the two functions can be used to determine a function's return address, global static pointer, and workspace pointer.

```
void p()
{
 typedef struct paramblock

   { void *return_address;
     void *gsb;
     int regparam1, regparam2;
   }
    paramblock;

  paramblock *pp = (paramblock *)_params;

  /* Return address is: pp->return_address
     global static base sb is: pp->gsb
     caller Wptr is: (void *) (pp + 1)  */
}
```

11.6 Fatal runtime errors

Errors are generated at severity level *Fatal* by the C runtime system when the program cannot be run. Such errors may occur at startup or during program execution.

The main causes of runtime errors in a program are summarised below.

- Insufficient memory at startup.
- Stack overflow during execution.
- Illegal conditions detected by the library functions **`free`**, **`realloc`**, **`ProcInit`**, and **`ProcPriPar`**. These errors are described in detail under the function descriptions in chapter 2 of the accompanying Reference Manual.

When runtime errors occur the program terminates immediately with an error message. All runtime error messages are prefixed with '**`Fatal-C_Library`**'.

11.6.1 Runtime error messages

Fatal-C_Library-Out of memory in system startup [*number*]

This error is generated when insufficient static or heap space is available to run the program. *number* can take the following values:

1 – Insufficient static area in programs which incorporate mixed language code from previous 3L compiler toolsets.

2 – Insufficient static area in programs written using the current TCOFF-based toolsets.

3 – Insufficient heap space for the input and output channel arrays.

4 – Insufficient heap space for command line parameters to the program.

If this error occurs then either the available memory can be increased or the program recoded in a less memory-intensive way.

Fatal-C_Library-Stack overflow

This message is only generated when stack checking is enabled in the compiler. It indicates stack overflow in the program and may be remedied by increasing increasing the specified stack size. If no stack size has

been specified and the default has been assumed by the program then the stack size cannot be increased and the program should be recoded.

Fatal-C_Library-Error in free(), bad pointer or heap corrupted

This error indicates an invalid pointer passed to **free** or corruption of the heap. No specific recovery is possible and the program should be debugged.

Fatal-C_Library-Error in realloc(), bad pointer or heap corrupted

This error indicates an invalid pointer passed to **realloc** or corruption of the heap. No specific recovery is possible and the program should be debugged.

Fatal-C_Library-Incorrect allocation of process workspace

This error indicates that process workspace was not allocated from the heap. It is generated by **ProcInit** when an attempt is made to use process workspace which has not been allocated by the standard functions **malloc**, **calloc**, and **realloc**, which allocate space from the heap.

Fatal-C_Library-Nested Pri Pars are illegal

This error is generated by **ProcPriPar** when it is called from a high priority process. Calling **ProcPriPar** from a high priority process is prohibited in ANSI C.

11.7 Transputer in-line code

ANSI C provides a detailed mechanism for incorporating transputer assembly code inserts into C programs. The system uses the special keyword **__asm** which can be used to enclose sequences of transputer instructions.

The **__asm** statement and how to use it is described in chapter 4 of the accompanying Reference Manual.

11.8 Compiler diagnostics

This section lists diagnostic error messages generated by **icc**. The section is introduced by descriptions of some standard terms which may be encountered in the message texts.

11.8.1 Message format

Diagnostic messages are displayed in the standard toolset format for error messages. Details of the standard can be found in section A.6.1.

11.8.2 Severities

Diagnostics are tagged with a severity level which indicates their effect on the compilation. Severity levels are the same as those used in the toolset standard but have slightly different meanings, which are described below.

Warning severity diagnostics are generated whenever legal, but unorthodox programming styles are detected. Compilation is unaffected and object code is generated normally.

Error severity diagnostics are generated whenever the compiler detects a programming error from which it can recover. Compilation continues, but may abort if more errors are detected subsequently. No object code is generated.

Serious diagnostic messages are generated when programming errors are detected from which the compiler cannot recover. Compilation continues but code has been lost. No output is generated.

Fatal diagnostic messages are generated for the most serious syntactical errors and cause the compiler to discontinue processing immediately. However, they do not indicate failure of the compiler and should not be reported to INMOS. No output is generated.

Error, *Serious*, and *Fatal* diagnostic messages return error codes for handling by system MAKE programs and batch files.

11.8.3 Standard terms

This section explains some of the standard terms and notation used in compiler error messages.

abstract declarator

> When using explicit casts or when passing an argument to `sizeof()`, a data type must be specified. This can be done by declaring an object of the correct type without specifying the name of the object. Declarations of this type are called abstract declarations, because they apply to no known object.

Examples of abstract declarations are:

```
(int) a = b;        /* 'int' is the abstract
                       declarator */

sizeof(int [3]);    /* 'int [3]' is the abstract
                       declarator */
```

char

Stands for a single ASCII character.

context

Stands for a type, for example, 'character constant', 'integer constant', and 'string constant'.

deprecated declaration

This means that a function declaration is incomplete. Declarations should specify the type of the function and the type of each formal parameter. If there are no parameters then the function type `void` should be specified.

expression

Stands for a C expression.

filename

A file name.

function prototype

A function declaration which usually precedes the function definition. It declares the function's type and the types of its parameters.

identifier

A C identifier, for example, a variable or function name.

initialiser

An initial value which is assigned to an object at the time of its declaration.

message string

The string which follows a compiler directive.

op

An operator. Valid operators include: "++" or "−−", "− >", "<<=", and the unary operators &, *, + and −.

quote_char

A quote character for the **#include** directive. This could be ", ', <, or >.

store class

A C storage class. Valid classes are **static** or **extern**.

string

Any string of ASCII characters.

struct/union

A variable of type **struct** or **union**.

type

A type identifier. Valid types are **int**, **char**, and **float**.

void context

This can occur at any point in a program where a value is not expected, for example, calling a function without using the returned number.

instruction

A transputer instruction, or a pseudo-instruction as accepted by the **_asm** construct.

11.8.4 ANSI trigraphs

The ANSI specification includes a number of three character sequences that can be used to represent certain ASCII characters that may not be present on all keyboards. These sequences, known as **trigraphs**, are used in compiler error

messages to stand for these characters.

ANSI standard trigraph sequences consist of a sequence of 2 question marks followed by a third character. A complete list of ANSI trigraphs is given in chapter 3 '*New features in ANSI C*' of the accompanying User Manual.

11.8.5 Warning diagnostics

#define macro *identifier* **defined but not used**

The named macro has been defined, but not referenced in the rest of the program. This message is only generated if specifically enabled by the '**FM**' compiler option.

'&' unnecessary for function or array *identifier*

A pointer to a function or array is implied by use of the name alone; the '&' operator is not required.

'int *identifier*()**' assumed – 'void' intended?**

A function was defined without specifying its type. The compiler assumes a function of type `int` if no type is specified.

identifier **has been defined; pragma ignored**

The function specified in the `IMS_nolink` pragma has already been defined with a static link.

identifier **has not been declared; pragma ignored**

The function specified in the `IMS_nolink` pragma has not yet been declared.

identifier **is not a function; pragma ignored**

The argument to the `IMS_nolink` pragma must be a function name.

identifier **multiply translated, this translation ignored**

The `IMS_translate` pragma has been applied to *identifier* more than once.

identifier **has already been translated to** *string*

The `IMS_translate` pragma has been used to translate more than one name to string.

number **treated as** *number*`UL` **in 32-bit implementation**

No type was specified for the number. The compiler assumes `unsigned long` if no type was specified.

op: **cast between function and object pointer**

The specified operator has been used in an expression involving pointers of different types, that is, a function pointer and an object pointer (a pointer to an area in memory).

op: **cast between function and non-function object**

The operation is performed upon two arguments, one of which is a function, and the other an object.

type identifier **declared but not used**

The named identifier has been declared, but not used in the program.

actual type *type* **mismatches format '%***char***'**

The type of an argument to `printf` or `scanf` does not match that implied by the control string.

ANSI '*char char char***' trigraph for '***char***' found – was this intended?**

The specified three character sequence was found in the source program. This has been treated as an ANSI trigraph and substituted for the character shown.

argument and old-style parameter mismatch: *expression*

There is an old (non-prototype) style function definition in scope, and the type of an argument (after default argument promotion has taken place) does not agree with the type of the corresponding formal parameter.

Cannot generate stack check for *function* **(pragma nolink applied)**

A stack check requires a static link, and the function *function* has been specified not to receive a static link (using `IMS_nolink`). `icc` compiles

the function with the stack check omitted.

character sequence /* inside comment

The start-of-comment character sequence was detected within a comment. Check that the previous comment was terminated correctly.

Dangling 'else' indicates possible error

Within nested `if ... else` constructs, there is some ambiguity as to which 'if' relates to which 'else'.

Deprecated declaration *identifier*() **– give arg types**

In the prototype declaration of the named function, the argument's names and/or their types were not specified.

division by zero: *op*

Division, or remainder, by zero, will cause overflow.

Expected string as second argument - pragma ignored

The second argument to the `IMS_translate` pragma must be a string literal.

extern 'main' needs to be 'int' function

In a declaration of `main()`, the function should always be declared as type `int`.

extern *identifier* **not declared in header**

All objects must be declared before use. This message is only generated if specifically enabled by the '`FM`' compiler option.

floating point constant overflow: *op*

Floating point overflow occurred during addition, subtraction, multiplication or division of two constants.

floating point overflow when folding

Floating point overflow occurred during addition, subtraction, multiplication or division of a constant.

floating to integral conversion failed

Conversion (casting) from a floating point type to an integral type (such as `int`) failed.

formal parameter *identifier* **not declared – 'int' assumed**

A formal parameter has been listed in the parameter list of the function definition, but there is no entry for it in the declaration list; it is therefore assumed to be of type `int`.

Format requires *count* **parameter(s), but** *count1* **given**

A call to `printf` or `scanf` was made with the incorrect number of arguments. The control string indicated that *count* arguments are needed, but *count1* were provided. This warning is only generated if pragma `IMS_on (pc)` is active. The header file `stdio.h` includes this pragma.

Illegal format conversion '%*char***'**

The character sequence '`%`*char*' is not a legitimate conversion character for `printf` or `scanf`. This warning is only generated if pragma `IMS_on (pc)` is active. The header file `stdio.h` includes this pragma.

implicit narrowing cast: *op*

The result of an operation performed at higher precision is immediately, and implicitly, cast to lower precision, thus losing the extra precision: if the extra precision is not required, the operation ought to be performed at the lower precision.

implicit return in non-void *identifier*()

The function does not contain a `return` statement, even though it is defined to return a value.

Incomplete format string

The control string for use with `printf` or `scanf` is incomplete. This warning is only generated if pragma `IMS_on (pc)` is active. The header file `stdio.h` includes this pragma.

inventing 'extern int *identifier*();'

No declaration exists for the function; it will be defined by default as `extern int`.

label *identifier* **was defined but not used**

The named label was set, but not used.

Linkage already set - pragma ignored

The **IMS_linkage** parameter has been specified more than once.

lower precision in wider context: *op*

The result of an operation performed at lower precision is immediately cast to a higher precision; it may be that the user was expecting the operation to be performed at the higher precision.

Missing comma in pragma argument list - pragma ignored

Multiple arguments to a pragma must be separated by commas.

no side effect in void context: *identifier*

The value which has been returned by an expression is not being used. This error would occur, for example, when a non-void function is called and the returned value is ignored.

non-portable – not 1 char in '...'

The characters enclosed by single quotes represent more than one character. The compiler will read the first character only, for example, 'AB' will be read as 'A'.

non-value return in non-void function

A function which should return a value has terminated without using a return statement or with a return statement that has no arguments. The value received from the function by the calling routine is undefined.

odd unsigned comparison with 0 : *op*

a $\geq$ comparison of an unsigned integer with zero, or a $\leq$ comparison of zero with an unsigned integer, is always true.

omitting trailing '\0' for char [*count***]**

The char array is fully occupied by characters and there is no room to append the string terminator (\0). *count* is the full length of the character array.

repeated definition of #define macro *identifier*

The named macro has been defined more than once. The definitions are identical.

Shift of *type* **by** *count* **undefined in ANSI C**

A shift of more than the number of bits in *type*, or less than zero was requested, undefined in ANSI C.

spurious {} around scalar initialiser

A scalar can take only one initialiser, so there is no need to use braces as are required with aggregate types such as arrays.

static *identifier* **declared but not used**

The named `static` object was declared but not used.

struct has no named member

A structure has been declared without any members.

Undefined macro *string* **in #if – treated as 0**

This error occurs when enumeration or undefined constants appear after the preprocessor `#if` directive. For example, if 'ab' and 'cd' are enumeration constants of the enumerated type 'abcd', the statement `#if ab == cd` would generate this error.

union has no named member

A union has been declared without any members.

unnamed bit field initialised to zero

A static declaration of a structure or union containing an unnamed bit field, the compiler has initialised that field to zero.

Unrecognised #pragma (no '(')
Unrecognised #pragma (no ')')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised #pragma *identifier*

identifier is not a pragma recognised by this compiler.

unused earlier static declaration of *identifier*

The static variable *identifier* has been defined before being declared. Generated only if the '**FH**' compiler option is specified.

use of *op* **in condition context**

Generated when the invalid operators '=' (assignment) or '~' (bit-not) are used in a condition statement.

variable *identifier* **declared but not used**

The variable was declared, but not used anywhere in the program.

(possible error): >= *number* **lines of macro arguments**

There are a suprisingly large number of lines of arguments to a macro; this may indicate a syntax error.

11.8.6 Recoverable errors

#error encountered *string*

The **#error** directive was found in the source code.

#*ident* **is not in ANSI-C**

#*ident* is not a recognised preprocessor directive.

context identifier **may not be function – assuming function pointer**

An attempt was made to use a function where it was not expected, typically when a function is included as a component within a structure.

instruction **may not have a size specified**

An **_asm** pseudo-instruction may not be explicitly sized.

',' (not ';') separates formal parameters

A semicolon has been used to separate the formal parameters in a func-

tion definition (as in Pascal) instead of a comma.

'register' attribute for *identifier* **ignored when address taken**

An attempt was made to take the address of a variable with 'register' storage class. The register attribute will be ignored allowing the address to be taken.

int op pointer **treated as** *int op* **(int)** *pointer*

The expression involving a integer and a pointer will result in the pointer being converted (cast) to an integer.

op: **implicit cast of** *type* **to 'int'**

A non-integer object has been used where an `int` was expected, for example, attempting to use a `double` as an argument to a switch statement (which requires an integer type).

op: **implicit cast of non-0 int to pointer**

Evaluation of the expression will result in the cast of an integer to a pointer.

op: **implicit cast of pointer to 'int'**

Evaluation of the expression will result in the cast of the pointer to an integer.

operator: **implicit cast of pointer to non-equal pointer**

Evaluation of the expression will result in the cast of one pointer type to another.

pointer operator int **treated as (int)** *pointer operator int*

Evaluation of the expression will result in the cast of the pointer to an integer.

Ancient form of initialisation, use '='

A }, rather than =, was used to introduce an initialiser, this is no longer legal C.

ANSI C does not support 'long float'

An object has been declared of type `long float`, this is illegal in ANSI C, which supports `float`, `double`, or `long double`.

Array of *type* illegal – assuming pointer

An array of functions or void objects has been declared. The compiler treats this as an array of pointers to functions or void objects.

Array [0] found

An empty array has been defined and will be set up instead as an array with one element.

assignment to 'const' object *identifier*

The expression contains an assignment to a constant. The assignment will be carried out.

const typedef *identifier* has const respecified

A typedef which is already qualified with `const`, has been qualified with `const`.

comparison *op* of pointer and int: literal 0 (for == and !=) is only legal case.

The specified operator was used to compare an object of type `int` and one of a type `pointer`. The only legal comparison of this type is between a pointer and 0 using either == or !=.

declaration with no effect

No name has been declared for the object. Specifying only the type of an object generates this error.

differing pointer types: *op*

The specified operator was used with pointers of different types.

differing pointer types: ':'

Types of objects in the conditional expression do not match.

Digit 8 or 9 found in octal number

A decimal digit was encountered in an octal number.

duplicate macro formal parameter: '*identifier*'

The function macro has two formal parameters with the same name.

duplicate member *identifier1* **of** *identifier2*

Two fields of structure or union *identifier2* have the name *identifier1*.

ellipsis (...) cannot be only parameter

A function declared to take a variable number of parameters must have at least one known parameter.

extern *identifier* **mismatches top-level declaration**

An `extern` declaration of *identifier* within a function definition does not match an `extern` declaration of *identifier* at the top level.

formal name missing in function DEFINITION

The type of a formal parameter has been omitted in a function definition.

function *identifier* **may not be initialised – assuming function pointer**

Initialisers cannot be used in function declarations or definitions.

function prototype formal *identifier* **needs type or class – 'int' assumed**

The type of a formal parameter has been omitted in a function declaration and `int` has been assumed.

hex number cannot have exponent

A hex number ending in `e` may not be immediately followed by + or −; separate the number and the additive operator with whitespace.

illegal bit field type *type* **– 'int' assumed**

Bit fields cannot be set within non integral variables. The compiler assumes an `int` instead.

illegal indirection on (void *): '*'

An attempt has been made to take the contents of the object pointed to by a pointer to void.

illegal option –D*identifier identifier*

The compiler D option must be specified for each assignment.

illegal string escape '*char**' – treated as** *char*

The character following \ does not form part of a valid string escape. The compiler treats the sequence *char* as *char.*

illegal [] member: *identifier*

An open array may not be a member of a structure or union.

implicit cast (to *type***) overflow**

Overflow occurred when casting an expression.

junk at end of #*identifier* **line – ignored**

The text following the directive is invalid and will be ignored.

linkage disagreement for *identifier* **– treated as** *store class*

The storage class of a previously defined **static** or **extern** object or function disagrees with the current declaration. The object will be treated as though it is in storage class *store class.*

L'...' needs exactly 1 wide character

A wide character constant should contain exactly one wide character.

Missing newline before EOF – inserted

A blank line should have been inserted before the end-of-file character.

Missing type specification – 'int' assumed

A type specification is missing. The object will be assumed to be of type **int**.

more than 4 chars in character constant

More than 4 ASCII characters were used to represent a character constant. When using the single quote syntax for character constants a maximum number of 4 characters is permitted in order to accommodate the octal representation of a character. The first 4 characters will be used.

no chars in character constant ''

No characters or character codes have been specified for the character constant. A NULL character is assumed.

number missing in #line

There is no line number following the preprocessor **`#line`** directive.

objects that have been cast are not l-values

An object that has been cast in l-value context; ANSI has made this illegal.

Omitted *type* before formal declarator – 'int' assumed

No type was specified; type **`int`** will be assumed.

operand of # not macro formal parameter: '*identifier*'

The operand to the # preprocessor operator must be a formal parameter of the function macro containing it.

overlarge escape '*number1*' treated as '*number2*'

An octal number in an escape sequence is too large to be represented in the target architecture.

overlarge escape '\x*number1*' treated as '\x*number2*'

A hexadecimal number in an escape sequence is too large to be represented in the target architecture.

parentheses (...) inserted around expression following *text*

Parentheses were expected after the specified text, for example, around a conditional expression such as an **`if`** statement.

prototype and old-style parameters mixed

It is illegal to mix new (prototype) and old-style parameter declarations.

return *expression* **illegal for void function**

A return statement was found within a void function. The return statement is ignored.

signed constant overflow: *op*

Overflow occurred when performing *op* upon signed, constant operands.

size of 'void' required – treated as 1

'void' was used as an argument to **sizeof**. The compiler assumes the size of void to be 1.

size of a [] array required, treated as [1]

The array is of unspecified size. In these circumstances **sizeof** return the size of the array type.

size of function required – treated as size of pointer

A function name was passed to the **sizeof** function. In these circumstances **sizeof** returns the size of the pointer to the function.

sizeof *bit field* **illegal – sizeof(int) assumed**

A bit field was passed to the **sizeof** function. In these circumstances **sizeof** casts the bit field to an integer and then returns its size.

Small (single precision) floating value converted to 0.0

The number is too small to represent in a single word (32 bit) floating point format, and has been rounded to 0.0.

Small floating point value converted to 0.0

The number is too small to represent in a double word (64 bit) floating point format, and has been rounded to 0.0.

Spurious #elif ignored

The **#elif** directive could not be matched with a corresponding **if**

directive and has been ignored.

Spurious #else ignored

The **`#else`** directive could not be matched with a corresponding **`if`** directive and has been ignored.

Spurious #endif ignored

The **`#endif`** directive could not be matched with a corresponding **`if`** directive and has been ignored.

struct member *identifier* **may not be function – assuming pointer**

A structure member was declared of function type; the compiler treats this as pointer to function type.

struct tag *identifier* **not defined**

A structure has been referenced before being defined.

type or class needed (except in function DEFINITION) – 'int' assumed

The type or storage class has been omitted from the function declaration.

Undeclared name, inventing 'extern int *identifier*'

An undeclared identifier was encountered and will be given the storage class **`extern`**.

union member *identifier* **may not be function – assuming pointer**

A union member was declared of function type; the compiler treats this as pointer to function type.

union tag *identifier* **not defined**

A union has been referenced before being defined.

unsigned constant overflow: *op*

Overflow occurred when performing *op* upon unsigned, constant operands.

unprintable char *number* **found - ignored**

An unprintable character was found in the source text.

volatile typedef *identifier* **has volatile respecified**

A typedef which is already qualified with `volatile`, has been qualified with `volatile`.

wrong number of parameters to *function*

A function was called with the wrong number of arguments.

11.8.7 Serious errors

op: **cast to non-equal** *type* **illegal**

A structure or union has been cast into a structure or union of a different type. The cast is illegal and will be ignored.

operator : **illegal cast of** *type* **to pointer**

A variable has been cast into a pointer type. The cast is illegal and will be ignored.

op: **illegal cast to** *type*

An illegal cast has been attempted. The cast is illegal and will be ignored.

context: **illegal use in pointer initialiser**

An object of type `auto`, or its address, cannot be initialised.

'...' must have exactly 3 dots

An ellipsis must consist of three dots.

'break' not in loop or switch – ignored

A break statement was encountered outside the scope of a loop or switch statement. A break at this point is illegal and will be ignored.

'case' not in switch – ignored

A case prefix has been encountered outside the body of a switch state-

ment. A case statement at this point is illegal and will be ignored.

'continue' not in loop – ignored

A continue statement has been encountered outside the body of a loop. A continue statement at this point is illegal and will be ignored.

'default' not in switch – ignored

A default prefix has been encountered outside the body of a switch statement. A default prefix at this point is illegal and will be ignored.

'goto' not followed by label – ignored

The text following a goto statement does not represent a label.

'void' values may not be arguments

Formal parameters in function definitions or declaration cannot be of type void.

'while' expected after 'do' – found *text*

The **while** statement is missing from a **do ... while** construct. *text* marks the position.

'{' of function body expected – found *text*

The opening brace in the body of a function is missing.

'{' or *identifier* **expected after** *type*, **but found** *text*

The opening brace following a **struct**, **union** or **enum** is missing. *text* marks the position.

':' expected but found a *symbol*

A label definition inside an **_asm** construct was not terminated by a colon.

<asm-directive> expected but found a *text*

text indicates where the **_asm** directive was expected.

<command> expected but found a *text*

Statements such as **switch** or **if** should be followed by a command. *text* indicates where the command was expected.

<expression> expected but found *text*

text indicates where the expression was expected.

identifier **expected but found** *text* **in 'enum' definition**

Reserved words cannot appear in the definition of enumerated types.

function **has pragma nolink specified, but accesses static data**

The specified function has been specified not to receive a static link (via **IMS_nolink**), but attempts to use static data. It is only possible to use static data when a static link is available.

identifier **is not a label - ldlabeldiff ignored**

The operands to the **ldlabeldiff** pseudo-instruction must be labels.

instruction **not followed by label - ignored**

A load or store **_asm** instruction must have a constant or label operand.

op **may not have whitespace in it**

Two-character operators such as '+=' must not contain spaces.

store class **variables may not be initialised**

Some types of C variables, such as those declared as **extern**, cannot be initialised.

Array size *count* **illegal – 1 assumed**

Arrays cannot be larger than 0xffffff on a 32-bit target, or 65535 on a 16-bit target.

attempt to apply a non-function

A name not declared as a function has been used in a context where a function should be.

attempt to include struct/union *identifier* object/member

A structure or union declaration may not contain a field of the structure or union type, or a field which references another field.

bit fields do not have addresses

Elements of type bit field in C structures cannot be addressed.

Bit size *size* illegal – 1 assumed

Bit sizes greater than 32 are set to 1.

Cannot call *function* (it requires a static link)

An attempt has been made to call the specified function which requires a static link, from a function which has been specified not to receive a static link (via `IMS_nolink`).

Cannot call through *pointer* (it requires a static link)

An attempt has been made to call a function through the specified pointer from a function which has been specified not to receive a static link (via `IMS_nolink`). All calls through function pointers are assumed to require a static link.

Cannot store to *identifier*

identifier is a built-in name, such as `_lsb` or `_params`, which cannot be assigned to.

char and wide (L"...") strings do not concatenate

A char string and a wide char string appear adjacently in the source text. Normally, adjacent strings in the source text are concatenated; however, this is not possible here, as they have different types.

differing redefinition of #define macro *identifier*

The named macro has been defined more than once. The definitions are not identical.

Digit required after exponent marker

Exponents of floating point numbers must be followed by a numeric character. The numeric character may be preceded by '+' or '–'.

duplicate 'default' case ignored

The default prefix has already been specified for the switch construct. The original definition will be used.

duplicate definition of *identifier*

The named identifier has already been defined.

duplicate definition of *struct/union* **tag** *identifier*

The named structure or union identifier has already been used.

duplicate definition of label *identifier* **– ignored**

The specified identifier has already been used. The original definition will be used.

duplicate type specification of formal parameter *parameter*

The specified parameter has been listed more than once in the function's formal parameter list.

duplicated case constant: *constant*

The constant has been specified more than once in the same case statement.

EOF in comment

The end-of-file character was detected inside a comment.

EOF in string

The end-of-file character was detected within a string.

EOF in string escape

The character sequence '\EOF' was detected within a string.

EOF not newline after #if ...

An end-of-file character was found after the '`#if`' directive; a newline character was expected.

expected *symbol1* **– inserted before** *symbol2*

symbol1 was expected before *symbol2* and the compiler has changed the code accordingly. For example, in the code "`if (TRUE printf();`" the compiler would expect to find '`)`' before '`printf`'.

expected *symbol1* **or** *symbol2* **– inserted** *symbol1* **before** *symbol3*

symbol1 or *symbol2* was expected before *symbol3*, but neither was found. *symbol1* is suggested as the most appropriate choice and the compiler has changed the code accordingly.

Expected <identifier> after *operator* **but found** *text*

The specified operator must be followed by an identifier. This error may occur after the structure member operator '`.`' and the structure pointer operator '– >'.

Expecting <declarator> or <type>, but found *text*

An identifier or type was expected at *text*. For example, the declaration '`typedef int *[3] test;`' generates this error.

Grossly over-long floating point number

There are too many digits in the floating point number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long hexadecimal constant

There are too many digits in the hexadecimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long number

There are too many digits in the decimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Hex digit needed after 0x or 0X

The hexadecimal specifier `0x` must be followed by a valid hexadecimal digit. The compiler assumes a zero digit.

Identifier (*name***) found in** *abstract declarator* **– ignored**

An identifier should not be used in an abstract declarator. This error is

generated, for example, if `sizeof(int *test[3]);` is used instead of the correct form `sizeof(int *[3]);`.

Illegal in *context*: *error*

Illegal expressions such as those involving division by zero generate this error.

Illegal in *expression*: **non constant** *identifier*

A constant is required in certain expressions, for example after a `case` prefix.

Illegal in l-value: *context*

An l-value was expected. For example, attempting to assign a value to a constant will generate this error.

Illegal in l-value: 'enum' constant *identifier*

Enumeration constants cannot be used as l-values in an expression.

Illegal in lvalue: function or array *identifier*

Arrays and function declarators cannot be used as l-values. This error would be generated, for example, by attempting to assign a value to a function declarator.

Illegal in the context of an l-value: *op*

The operator *op* cannot appear in l-value context.

Illegal types for operands: *operator*

The operator has been used with an invalid type. For example, it is illegal to use the structure member operator '`.`' with a variable of type `int`.

Illegal 'void' member/object: *identifier*

An object or member of a structure or union cannot be declared as being of type `void`.

incomplete tentative declaration of *identifier*

The declaration of *identifier* has gone out of scope before the declaration has been completed.

Junk after #if *expression*

The **#if** directive must be terminated by a newline character.

Junk after #include *filename*

The **#include** directive must be terminated by a newline character.

label *identifier* **has not been set**

A label has been referenced but not set. This message will be generated if **goto** is used with an undefined label.

ldlabeldiff not followed by label - ignored

The operands to the **ldlabeldiff** pseudo-instruction must be labels.

Misplaced 'else' ignored

An **else** statement was found where it was not expected. It will be ignored.

Misplaced '{' at top level – ignoring block

An opening brace was found at the top level of a program when it was not expected, for example when not used as part of a function or structure definition.

Misplaced preprocessor character *char*

A preprocessor directive character (# or \) was found where it was not expected.

Missing #endif at EOF

An **#endif** directive is missing. This error will not be generated until the last of the currently open files is about to be closed (ANSI standard does not require **#if** and **#else** statements to match in included files).

Missing *quote char* **in preprocessor command line**

A 'quote' character is missing from a preprocessor command line. The missing character could be ', <, >, or ".

Missing ')' after *identifier* **(... on line** *number*

A closing parenthesis is missing from the macro which will be substituted at line *number*.

Missing ',' or ')' after #define *identifier* **(...**

The list of parameters in a macro definition is either incomplete or has not been correctly terminated by a closing parenthesis.

Missing < or " after #include

The opening 'quote' character which introduces the filename is missing.

Missing hex digit(s) after \x

The hexadecimal introducer sequence **\x** was found, but no hexadecimal digit was specified. The compiler assumes that the letter **x** was intended.

Missing identifier after #define

The definition is empty. **#define** must be followed by an identifier.

Missing identifier after #ifdef

#ifdef must be followed by an identifier.

Missing identifier after #undef

#undef must be followed by an identifier.

Missing parameter name in #define *identifier* **(...**

A parameter is missing from the specified macro definition. This error would be generated by a definition of the form **#define test(arg,)**.

Newline or end of file within string

A newline or end-of-file character was encountered within a string.

No ')' after #if defined(...

The closing parenthesis is missing from the directive.

No identifier after #if defined

`#if defined` must be followed by an identifier.

Non-formal *identifier* **in parameter-type-specifier**

The parameter *identifier* was included in the declarator list of a function, but not in the parameter list. For example, a definition such as `int foo() int x; {}` would generate this error.

non-static address *identifier* **in pointer initialiser**

Pointers cannot be initialised with the address of an object of type `auto`.

Number *number* **too large for 32-bit implementation**

The specified number is too large to be represented in 32 bits.

Operand *number* **to** *instruction* **is larger than a word**

The arguments to an `_asm` load or store pseudo-instruction must fit in a machine word.

Operand *number* **to** *instruction* **is not word-sized**

The arguments to an `_asm` store pseudo-instruction must fit exactly in a machine word.

Operand to *instruction* **must be a constant or local variable**

An illegal operand has been given to an `_asm ldl` or `stl` instruction.

Operand to *instruction* **is larger than a word**

The operand to a primary instruction inside `_asm` must fit in a machine word.

Overlarge (single precision) floating point value found

The number is too large to represent in single word (32 bit) floating point format.

Overlarge floating point value found

The number is too large to represent in double-word (64 bit) floating point format.

quote (*char*) inserted before newline

The specified quote character was found before a newline character. This may indicate a a spurious character or a missing closing quote.

re-using *struct/union* **tag** *identifier* **as** *union/struct* **tag**

The named identifier has been used to identify two different types of object.

size of expression unknown: treated as 0

The size of a structure or union is required, but the structure or union has not been completely declared.

size of struct *identifier* **needed but not yet defined**

The size of the structure has not yet been defined. This error can occur when an undefined structure is used as an argument to the `sizeof` function and when an undefined structure is used in the declaration of a variable. In the second case the error occurs because the compiler attempts to determine the size of the structure for memory allocation purposes.

static function *identifier* **not defined – treated as extern**

A function was defined as `static` in the function prototype, but the compiler was unable to find the function definition. An `extern` function is assumed.

storage class *store class* **incompatible with** *store class* **– ignored**

Two incompatible storage classes have been used in a declaration. For example, `extern static foo;` generates this error because `extern` and `static` are incompatible types.

storage class *store class* **not permitted in context** *context* **– ignored**

The specified storage class is not permitted in the context in which it has been used. This error would be generated, for example, if storage class `auto` were to be used at the top level.

string initialiser longer than char [*count***]**

A character array has been initialised with more characters than the array can accommodate. Since the compiler adds a terminating NULL charac-

ter to strings, string initialisers should always contain one less element than the array.

struct *identifier* **must be defined for (static) variable declaration**

An undefined structure has been used in a variable declaration.

struct/union *identifier* **has no** *identifier* **field**

The structure or union contains no field of that name.

struct/union *identifier* **not yet defined – cannot be selected from**

A reference was made to an undefined structure or union.

Too few arguments for *instruction*

A load or store `_asm` pseudo-instruction has too few arguments.

Too few arguments to macro *identifier*(... **on line** *number*

There are too few arguments to the macro which will be substituted at line *number*.

Too many arguments for *instruction*

A load or store `_asm` pseudo-instruction has too many arguments.

Too many arguments to macro *identifier*(... **on line** *number*

There are too many arguments to the macro which will be substituted at line *number*.

too many initialisers in {} for aggregate

An aggregate type, for example an array, has been initialised with more values than can be accommodated.

type *type1* **inconsistent with** *type2*

Two incompatible type identifiers are being used in the declaration of a single object. For example, the declaration `double int x;` would generate this error.

type disagreement for *identifier*

The specified identifier has already been assigned a different type.

typedef name *type* **used in expression context**

A type definition has been used in an expression.

undefined struct/union *identifier* **cannot be member**

The structure or union being used as a member of another structure or union is at present undefined.

undefined struct/union *identifier1* **member/object:** *identifier2*

The structure or union is, at present, undefined.

Uninitialised static [] arrays illegal

Static arrays of unspecified size must be initialised.

union *identifier* **must be defined for (static) variable declaration**

An undefined union has been used in a variable declaration.

Unknown directive: #*identifier*

identifier is not a valid preprocessor directive. Check spelling and/or syntax.

*****space* and ***tab* are invalid string escapes**

Whitespace ('*space*' or '*tab*') was found within a string. All characters up to the first non-whitespace character are ignored; if the first non-whitespace character is a newline character, this will also be ignored.

{} must have 1 element to initialise scalar or auto

When initialising a scalar quantity or `auto` variable only one initialiser should be specified within the enclosing braces.

first or last token in #define body

The ## preprocessor operator must be preceded by a preprocessor token, and succeeded by a preprocessor token.

11.8.8 Fatal Errors

#error encountered *string*

The **#error** directive was found. This directive normally causes the compilation to abort immediately but can be disabled by the **IMS_off(ef)** pragma. The message is still generated.

11.9 icc error messages

This section lists messages generated by **icc** on encountering file system and other errors. Errors are listed by severity level.

icc generates errors at three severity levels: **Warning**; **Serious**; and **Fatal**.

11.9.1 Warnings

Cannot delete temporary file *filename*

Host file system error.

Expression generates poor code on this target ('dup' required)

An expression is being compiled for a tranputer class of which only some members have the *dup* instruction. The compiler has decided that the expression could be compiled more efficiently using the *dup* instruction, but cannot do so because it is not present on all members of the class.

Floating-point generates poor code on this target

Floating-point code is being compiled for a transputer class of which only some members have instruction set additions to enhance floating-point performance. As these instructions are not present on all members of the class, the compiler cannot use them.

Too many compiler arguments
Too many assembler arguments

There are too many options on the command line. The extra options are ignored.

11.9.2 Serious errors

#include file *filename* wouldn't open

The file *filename* could not be opened.

illegal character (*number* = '*char*') in source
illegal character (hex code *number*) in source

An unexpected character was found in the source code. The ASCII code of the character (if printable), and the character itself, are given.

11.9.3 Fatal Errors

Invalid command line option (*text*)

text is not a recognised command line option.

Invalid source file name (*filename*)

filename is not a valid source file name. (Source file names may not contain hyphens.)

I/O error writing *filename*

An error occurred when writing to the named file.

Missing include directory name

The J command line option must be followed by a directory name.

Missing object file name

The O command line option must be followed by an object file name.

No file name given

No source file was specified on the command line.

Out of memory
Out of store (for error buffer)
Out of store (in cc_alloc)

The compiler ran out of available memory.

Too many errors

After 100 Serious errors, the compilation aborts.

12 icconf – configurer

This chapter describes the configurer tool **icconf** that configures code for transputer networks. It describes the command line syntax and explains how the tool is used to generate a configuration data file for input to the code collector tool. The chapter ends with a list of error messages.

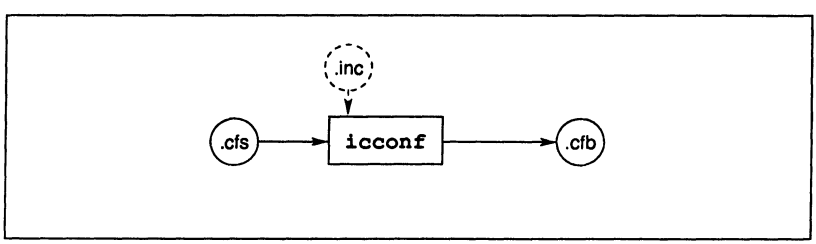
12.1 Introduction

The configurer takes a configuration description created using the transputer configuration language and produces a configuration data file which **icollect** uses to generate bootable code for a transputer network.

A configuration description describes how code is to be run on a network of transputers. It consists of separate definitions of the software and hardware networks, and a mapping description which defines how the software will be placed on the processor network. Using this description the configurer allocates code to particular processors and performs wide ranging consistency checks on the mapping of software to hardware.

Code to be run on separate processors must be linked code. Linked units that are to be run on the same transputer must be compiled for the same or a compatible transputer type.

The operation of the configurer tool in terms of toolset default file extensions is illustrated below.



12.2 Configuration language implementation

The configuration language supported by **icconf** has a number of implementation characteristics of which the programmer should be aware. These are briefly listed below; details can be found in section C.1.

- Array subscript ranges are machine word-length dependent.

- Source lines must not exceed 512 characters. Leading and following white space is ignored.
- Dimensions for symbols and array constants must not exceed 16.
- The number of characters in external symbol names in a linked object file must not exceed 256.

12.3 Running the configurer

The configurer takes as input a configuration description file and produces an object data file for input to the collector tool.

To run the configurer use the following command line:

▶ **icconf** *filename* {*options*}

where: *filename* is the configuration description file. The filename is interpreted as given and no file extension is assumed. Only one file may be specified.

options is a list of one or more options from table 12.1.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Examples of use:

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icconf hello.cfs
icollect hello.cfb -t
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icconf hello.cfs
icollect hello.cfb /t
iserver /sb hello.btl /se
```

Option	Description
C	Checks the configuration description. No configuration data file is generated.
G	This option is used when debugging and disables any ordering of process memory segments in the configuration code.
I	Displays extra information as the tool runs.
L	Loads the tool onto the transputer board and terminates.
O *filename*	Specifies an output filename. If no output file is specified the configuration data file is given the base name of the input file and the `.cfb` extension is added.
P *procname*	Specifies the name of the root processor when configuring for EPROMs. *procname* must not be an element from an array of processors.
RA	Creates a file suitable for a boot-from-ROM application in which the code and data are both loaded into RAM.
RO	Creates a file suitable for a boot-from-ROM application in which the code is loaded into ROM and the data is loaded into RAM.
RS *romsize*	Specifies the size of ROM on the root processor. Only valid when used with the 'RA' or 'RO' options. *romsize* is specified in decimal format and can be followed by 'K' or 'M' to indicate kilobytes or megabytes.
W	Disables configurer messages of severity *Warning*.
WP	Generates additional *Warning* messages.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

Table 12.1 Configurer options

12.3.1 Default command line parameters

Default command line parameters can be defined on the system in the `ICCONFARG` environment variable. Parameters must be specified in the variable using the syntax required by the configurer command line.

12.3.2 Boot-from-ROM options

The boot-from-ROM options 'RO' and 'RA' indicate that the program is to be collected for loading into EPROM and select the execution mode (from ROM or RAM) for the root transputer code.

Note: The same boot-from-ROM option ('RO' or 'RA' as appropriate) *must* also be supplied to `icollect` when the EPROM-loadable program is created. The option specifies to the collector the correct EPROM mode for the program.

For further details see section 13.2.5.

12.3.3 Standard include files

A number of standard include files are supplied to assist with configuration.

Configurer defaults are defined in the file `setconf.inc` which is read by the configurer at startup. This file is automatically included and does not need to referenced by an `#include` statement. `setconf.inc` contains a number of boolean constants required by `icconf`, definitions of configuration base types, and predefined INMOS transputer types.

Other standard include files provided with the toolset need to be included in the normal way. They provide definitions of processor and memory combinations for INMOS *i*q systems systems products and a number of configuration examples that can be used as templates for configuration descriptions.

All standard configuration include files supplied with the toolset carry the `.inc` extension. The files supplied are listed below.

`trams.inc`	`processor` type definitions for INMOS *i*q systems TRAnsputer Modules (TRAMs).
`boards.inc`	Processor type definitions for INMOS *i*q systems transputer evaluation boards.

12.3.4 Configuration description examples

A series of example configuration descriptions is supplied on the `config` examples subdirectory.

There are a number of files provided on this subdirectory including a configuration description for the 'Hello world' program used in chapter 3, and a series of configurations for specific network topologies such as rings, grids, trees, and pipelines.

12.3.5 Configurer library files

The configurer reads a special library file at startup which contains the system startup processes for different transputer types and error modes. The file is called **sysproc.lib** and is searched for on the directory specified by **ISEARCH**. This is normally the toolset **libs** directory, in which the file was originally installed.

12.3.6 Search paths

If a directory path is not specified the configurer uses the standard toolset search mechanism for locating input files, include files, and system library files. Briefly, the current directory is searched first, followed by the directories specified by **ISEARCH** (if defined on the system). For details see section A.3.

12.3.7 Default memory map

By default the configurer places code into memory in the following order beginning at **LoadStart** (a location above MemStart defined by **icollect**): stack; code; vector space; static; heap; system data; system code; and freespace. The memory segments are contiguous.

The default memory map is illustrated in figure 12.1.

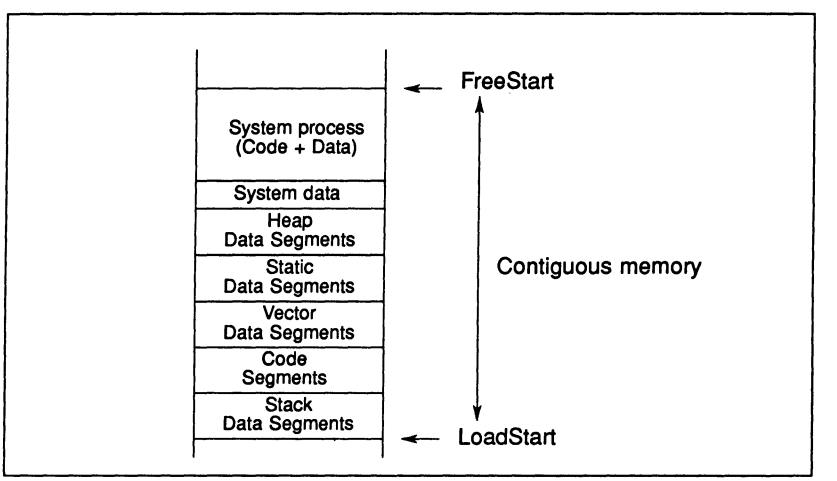


Figure 12.1 **icconf** default memory map

12.4 Configurer diagnostics

Errors in the configuration source produce diagnostic messages in standard toolset format. Details of the format can be found in section A.6.1.

Diagnostics are generated at severities *Warning*, *Error*, and *Serious*. No diagnostics are generated at severity level *Fatal*. The configurer aborts after 200 source file errors have been detected.

Diagnostic messages are listed in the following sections by severity.

Note: The following lists do not describe general **icconf** system errors, which are listed separately in section 12.5.

12.4.1 Warning messages

The following diagnostic messages are generated at severity level *Warning*.

attribute '*name*' definition ignored

This message can only occur in mixed language programs incorporating occam modules. The named **stacksize** or **heapsize** attribute has been assigned a value that has been ignored.

attribute '*name*' has been reassigned

Named attribute has been reassigned and can occur for any attribute.

attribute '*name*' undefined

Named attribute has not been assigned a value.

channel '*name*' unconnected and unplaced

Named channel has not been connected or placed.

connector '*name*' unused

Named connector has not been used in a connect statement.

link '*name*' unconnected

Named link has not been connected.

nested comment statements, *value*

One or more nested comments have been found by the configurer. *value* is the number of nested comments found.

order attributes ignored in debug mode

The G option has been specified and the memory segments of one or more user process has been given an order priority in the configuration source.

overflow in hexadecimal escape character

A numerical overflow has occurred during the evaluation of a hexadecimal escape character whose range is from 0 to 255.

overflow in octal escape character

A numerical overflow has occurred during the evaluation of an octal escape character whose range is from 0 to 255.

processor '*name***' unconnected**

Named processor has not been connected to the network (defined by the tree of connections from the root processor).

processor '*name***' unused**

Named process has been connected to the network but has had no user processes placed onto it.

12.4.2 Error messages

The following diagnostic messages are generated at severity level **`Error`**.

attribute '*name***' cannot be reassigned**

Named attribute cannot be reassigned. This can only occur with the **`element`** and **`type`** attributes of nodes and processors.

attribute '*name***' multiply defined in '***name***'**

Named attribute has been declared within the **`interface`** attribute and its name clashes with a previously declared attribute or its name clashes with the name of a predefined attribute for the named symbol.

attribute '*name*' undefined in '*name*'

Named attribute is an undefined attribute of the named symbol.

attribute '*name*' undefined

Named attribute has not been assigned a value which is required.

channel '*name*' connected and unplaced

Named channel has been connected to an input (or output) edge and has not been placed onto a link.

channel '*name*' multiply connected

Named channel has been used more than once in a connect statement.

channel '*name*' multiply placed

Named channel has been used more than once in a place statement.

channel '*name*' unconnected and placed

Named channel has been placed and is unconnected.

connect '*name*' to '*name*' illegal, both edges

Connect statement is illegal because the named elements are both edges.

connect '*name*' to '*name*' illegal, both non-edges

Connect statement is illegal because the named elements are channels and they communicate in the same direction.

connect '*name*' to '*name*' illegal, edge/non-edge

Connect statement is illegal because the first named element is an input (or output) edge and the second named element is a channel and they communicate in different directions.

connect '*name*' to '*name*' illegal, non-edge/edge

Connect statement is illegal because the first named element is a channel and the second named element is an input (or output) edge and they communicate in different directions.

connector '*name*' multiply placed

Named connector has been used more than once in a place statement.

connector '*name*' multiply used

Named connector has been used more than once in a connect statement.

constant dimension sizes inconsistent, *value*

A constant array has been defined which has inconsistent dimension sizes for some of its elements. *value* is the number of the incorrect dimension, counting from zero.

constant dimensions incompatible with '*name*'

Named symbol has been assigned a constant value whose dimensions do not match the symbol in number and/or size.

constant element types not equal, *type*

A constant array has been defined where some or all of its elements have non-equal types. *type* is the correct type for each element in the array.

constant type incompatible with '*name*', *type*

Named symbol has been assigned a constant value whose type is not the same as itself. *type* is the correct type for the constant.

element '*name*' in connection undefined

Named symbol has been used in a connect statement and has no associated data, for example attributes that are undefined, link attributes of processor types, channel attributes of process types etc.

element '*name*' in placement undefined

Named symbol has been used in a place statement and has no associated data, for example attributes that are undefined, link attributes of processor types, channel attributes of process types etc.

element '*name*' not completely subscripted

Named symbol has been defined as an array and has not been completely subscripted.

host edge '*name*' undefined

When configuring to boot from link, the named host edge has not been declared in the configuration source. This error can only be caused if the standard include file `setconf.inc` has been altered.

illegal RAM + ROM memory size for '*name*', *value*

Named symbol is a processor whose total RAM and ROM memory sizes exceed the total memory addressing capabilities for the processor. *value* is the amount of extra memory specified for the processor.

illegal RAM memory size for '*name*', *value*

Named symbol is a processor whose total RAM memory size exceeds the total memory addressing capabilities for the processor. *value* is the amount of extra memory specified for the processor.

illegal ROM memory size for '*name*', *value*

Named symbol is a processor whose total ROM memory size exceeds the total memory addressing capabilities for the processor. *value* is the amount of extra memory specified for the processor.

illegal assignment for attribute '*name*'

Named attribute is the wrong type of attribute for the assignment of a constant value, for example, the `order` attribute of a process.

illegal dimension size, *value*

A dimension size not greater than zero has been specified. *value* is the dimension number with the illegal dimension size.

illegal escape character sequence, *char*

An illegal escape character sequence has been specified. *char* is the illegal escape character.

illegal format character constant, *char*

An illegal format character constant has been specified. *char* is the unexpected character found in the character constant.

illegal format hexadecimal constant, *char*

An illegal format hexadecimal constant has been specified. *char* is the unexpected character found in the hexadecimal constant.

illegal number of dimensions for '*name***'**

Named symbol has too many dimensions, should have zero dimensions.

illegal number of dimensions, *value*

Number of dimensions for a symbol or constant exceeds the maximum number of dimensions allowed by the configurer. *value* is the maximum number of dimensions allowed.

illegal number of subscripts for '*name***',** *value*

Number of subscripts specified for the named symbol exceeds the number the symbol requires. *value* is the maximum number of subscripts allowed.

illegal number of subscripts for constant, *value*

Number of subscripts specified for a constant exceeds the number the constant requires. *value* is the maximum number of subscripts allowed.

illegal operation for attribute '*name***'**

Named attribute has been used incorrectly. That is, the use of the attribute does not conform to its syntactic specification, for example, using the node attribute `element` to form a list of parameters.

illegal source file character, *value*

An unexpected character has been found in the source file. *value* is the ASCII value for the illegal character.

illegal token for expression, found *token*

An unexpected token has been found at the start of an expression. *token* is the unexpected token.

illegal token for statement, found *token*

An unexpected token has been found at the start of a statement. *token* is the unexpected token.

Illegal type for '*name*' in USE statement, *type*

Named symbol has been specified in a use statement and is not a process or a process type. *type* is the type of the symbol.

Illegal type for '*name*' in connection, *type*

Named symbol has been specified in a connect statement and is not a channel, link or connector. *type* is the type of the symbol.

Illegal type for '*name*' in definition, *type*

Named symbol has been specified in a node definition statement and is not a node type. *type* is the type of the symbol.

Illegal type for '*name*' in expression, *type*

Named symbol has been specified in an expression and is not a constant value. *type* is the type of the symbol.

Illegal type for '*name*' in modification, *type*

Named symbol has been specified in an attribute modification statement and is not a node. *type* is the type of the symbol.

Illegal type for '*name*' in placement, *type*

Named symbol has been specified in a place statement and is not a process, processor, channel, link or connector. *type* is the type of the symbol.

Illegal type for '*name*', *type*

Named symbol does not have the type expected by the configurer. This will only occur if the name specified using the `P` option is not a processor or if the host edge `host` is in fact not an edge. *type* is the type of the symbol.

Illegal type for IF statement condition, *type*

The condition value for an if statement is not of integral type. *type* is the type of the condition value.

illegal type for arithmetic operator *operator, type*

The operand of an arithmetic unary operator is not of arithmetic type. *type* is the type of the operand and *operator* is the arithmetic operator.

illegal type for boolean operator *operator, type*

The operand of a boolean binary operator is not of integral type. *type* is the type of the operand and *operator* is the boolean operator.

illegal type for condition operator *operator, type*

The condition value for a conditional ternary operator is not of integral type. *type* is the type of the condition value and *operator* is the conditional operator.

illegal type for connector '*name***' in placement,** *type*

Named symbol is a connector defining a connection and has been used in the incorrect position in a place statement. For example, if the symbol describes a connection between channels then the symbol has been used after the **on** in the place statement. *type* is the type of connection defined by the symbol.

illegal type for dimension size, *type*

The type of a dimension size value is not of integral type. *type* is the type of the dimension size value.

illegal type for integral operator *operator, type*

The operand of an integral unary operator is not of integral type. *type* is the type of the operand and *operator* is the integral operator.

illegal type for subscript value, *type*

The type of a subscript value is not of integral type. *type* is the type of the subscript value.

illegal type for value in REP statement, *type*

The base or limit value for a replicator statement is not of integral type. *type* is the type of the base or limit value.

illegal types for arithmetic operator *operator*, *type1* **and** *type2*

The operands of an arithmetic binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the arithmetic operator.

illegal types for equality operator *operator*, *type1* **and** *type2*

The operands of an equality binary operator are not both of arithmetic type. *type1* and *type2* are the types of the operands and *operator* is the equality operator.

illegal types for integral operator *operator*, *type1* **and** *type2*

The operands of an integral binary operator are not both of integral type. *type1* and *type2* are the types of the operands and *operator* is the integral operator.

illegal use of constant for element

A constant value has been used in a situation where it is not required.

illegal use of subfield operator for '*name***'**

Named symbol which no associated subfields (or attributes) has been accessed using the subfield operator.

illegal use of subfield operator for constant

A constant value has been accessed using the subfield operator.

illegal value for attribute '*name***' in PRI PAR process**

This message can only be generated in mixed language programs incorporating occam modules. A named attribute has been given a value which is inconsistent with the type of the process it is associated with, that is, a high priority occam process has been specified to start in high priority.

illegal value for attribute '*name***'**

Named attribute has been given a value which is inconsistent with the type of the attribute and its semantic meaning, for example, assigning an integer value to the `type` attribute of a processor.

incompatible interface, '*name*' has different type, *type*
incompatible interface, '*name*' has too few parameters
incompatible interface, '*name*' has too many parameters
incompatible interface, '*name*' has unequal dimensions

These messages can only be generated in mixed language programs incorporating occam modules. The named symbol is an occam process and the **interface** defined for the process mismatches the formal parameter list defined in its associated object file.

insufficient RAM memory for '*name*', *value*

Named processor's total RAM memory size is insufficient for the number of processes placed on the processor (which includes their data requirements). *value* is the number of extra bytes needed to accommodate all the processes on the processor.

insufficient ROM memory for '*name*', *value*

Named processor is the root processor in a boot from ROM system and its total ROM memory size is insufficient for the number of processes placed on the processor. *value* is the number of extra bytes needed to accommodate all the processes on the processor.

link '*name*' multiply connected

Named link has been used more than once in a connect statement.

link '*name*' multiply placed

Named link has been used more than once in a place statement.

links '*name*' and '*name*' unconnected and placed

Named links are not connected to each other and have each been placed with channels which are connected to each other.

missing (for SIZE operator, found *token*

The size operator has been found and an opening parenthesis was expected to be found after the keyword **size**, instead of which the token *token* was found.

missing) for SIZE operator, found *token*

The size operator has been found and a closing parenthesis was expected to be found after the operand to the operator, instead of which the token *token* was found.

missing) for attribute list, found *token*

An attribute list has been found and a closing parenthesis was expected to terminate the list, instead of which the token *token* was found.

missing) for cast operator, found *token*

A casting operator has been found and a closing parenthesis was expected to be found after the type identifier, instead of which the token *token* was found.

missing) for expression, found *token*

A parenthesised expression has been found and a closing parenthesis was expected to be found after the sub-expression, instead of which the token *token* was found.

missing , or TO for CONNECT statement, found *token*

A connect statement has been found and a comma or the keyword `to` were expected to be found, instead of which the token *token* was found.

missing : for conditional operator, found *token*

A conditional operator has been found and a colon was expected to be found after the first sub-expression, instead of which the token *token* was found.

missing ; for statement, found *token*

A statement has been found which expects a semicolon to terminate it, instead of which the token *token* was found.

missing = for REP statement, found *token*

A replicator statement has been found and an equals was expected to be found after the replicator identifier, instead of which the token *token* was found.

missing = or (for attribute, found *token*

An attribute definition has been found and an equals or opening parenthesis were expected to be found after the attribute identifier, instead of which the token *token* was found.

missing FOR for USE statement, found *token*

A use statement has been found and the keyword `for` was expected to be found, instead of which the token *token* was found.

missing INCLUDE for # statement, found *token*

A hash has been found and the keyword `include` was expected to be found, instead of which the token *token* was found.

missing ON for PLACE statement, found *token*

A place statement has been found and the keyword `on` was expected to be found, instead of which the token *token* was found.

missing TO or FOR for REP statement, found *token*

A replicator statement has been found and the keywords `to` or `for` were expected to be found, instead of which the token *token* was found.

missing] for subscript, found *token*

A subscript operator has been found and a closing square bracket was expected to be found after the subscript value, instead of which the token *token* was found.

missing attributes for attribute list

An attribute list has been found which is empty.

missing constants for constant list

A constant list has been found which is empty.

missing identifier for DEFINE statement, found *token*

A define statement has been found and an identifier was expected to be found after the attribute list (if specified), instead of which the token *token* was found.

missing identifier for REP statement, found *token*

A replicator statement has been found and an identifier was expected to be found after the keyword **rep**, instead of which the token *token* was found.

missing identifier for VAL statement, found *token*

A value statement has been found and an identifier was expected to be found after the keyword **val**, instead of which the token *token* was found.

missing identifier for attribute list, found *token*

An attribute list has been found and an identifier was expected to be found after the opening parenthesis starting the list, instead of which the token *token* was found.

missing identifier for attribute, found *token*

An attribute list has been found and an identifier was expected to be found after a comma in the attribute list, instead of which the token *token* was found.

missing identifier for name, found *token*

A name expression has been found and an identifier was expected to be found at the start of the expression, instead of which the token *token* was found.

missing identifier for subfield, found *token*

A subfield expression has been found and an identifier was expected to be found after the subfield operator, instead of which the token *token* was found.

missing statements for statement list

A statement list has been found which is empty.

missing string for INCLUDE statement, found *token*

An include statement has been found and a string was expected to be found after the keyword **include**, instead of which the token *token* was found.

missing string for USE statement, found *token*

A use statement has been found and a string was expected to be found after the **use** keyword, instead of which the token *token* was found.

missing type for DEFINE statement, found *token*

A define statement has been found and a type identifier was expected to be found after the keyword **define**, instead of which the token *token* was found.

missing type for attribute, found *token*

An parameter list has been found and a parameter type was expected to be found after a comma in the parameter list, instead of which the token *token* was found.

missing } for constant list, found *token*

A constant list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

missing } for statement list, found *token*

A statement list has been found and a closing brace was expected to terminate the list, instead of which the token *token* was found.

modification of '*name***' illegal, already used**

Named symbol is a type identifier which has derived other symbols and an attempt has been made to modify of one of its attributes.

object file for '*name***' undefined**

Named process has been specified more than once in a use statement.

overflow in REP statement expression

A numerical overflow has occurred during the evaluation of a replicator statement, that is, the replicator identifier has overflowed.

overflow in arithmetic expression

A numerical overflow has occurred during the evaluation of an arithmetic expression.

overflow in decimal integer constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit decimal integer constant.

overflow in dimension size expression

A numerical overflow has occurred during the evaluation of a dimension size expression (which is done to the precision of the hosts integer word length).

overflow in dimension sizes for '*name*'

A numerical overflow has occurred during the evaluation of the number of elements of the named symbol (which is done to the precision of the host system's integer word length).

overflow in dimension sizes for constant

A numerical overflow has occurred during the evaluation of the number of elements of a constant array (which is done to the precision of the hosts integer word length).

overflow in hexadecimal integer constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit hexadecimal integer constant.

overflow in octal integer constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit octal integer constant.

overflow in real double constant

A numerical overflow has occurred during the conversion of a string representing a 64 bit real constant.

overflow in real float constant

A numerical overflow has occurred during the conversion of a string representing a 32 bit real constant.

overflow in subscript value expression

A numerical overflow has occurred during the evaluation of a subscript value expression (which is done to the precision of the hosts integer word length).

place '*name*' on '*name*' illegal, edge/non-edge

Place statement is illegal because the first named element is an input or output edge and the second named element is a link.

place '*name*' on '*name*' illegal, non-edge/edge

Place statement is illegal because the first named element is a channel and the second named element is an edge.

process '*name*' and channel '*name*' placed on different processors

Named process has been placed on a different processor than the named channel. That is, the named channel, which is one of the channels of the process, has been placed on the link of a different processor.

process '*name*' and processor '*name*' error modes mismatch

Named process has an error mode (defined by the object file associated with the process by the use statement) which is incompatible with other processes executing on the named processor.

process '*name*' and processor '*name*' target types mismatch

Named process has a target transputer type (defined by the object file associated with the process by the use statement) which is incompatible with transputer type of the named processor.

process '*name*' multiply USEd

Named process has been used more than once in a use statement.

process '*name*' multiply placed

Named process has been used more than once in a place statement.

process '*name*' unplaced

Named process has not been placed.

process type '*name*' multiply USEd

Named process type has been used more than once in a use statement.

processes '*name*' and '*name*' placed on different processors

Named processes (which are connected by channels) have been placed on different processes such that there is no unplaced link connection between the processors.

processor '*name*' unconnected and placed

Named processor has not been connected to the network and has had one or more user process placed onto it.

reference to undefined symbol '*name*'

Named symbol has been referenced but had not been defined at the point of reference.

root processor '*name*' undefined

When configuring to boot from ROM, the named processor (specified using the `P` option) has not been defined in the configuration source.

subscript out of range for '*name*', *value*

Named symbol has been accessed with the subscript operator and the subscript value used is outside the valid range for the symbol. *value* is the dimension number that was subscripted.

subscript out of range for constant, *value*

A constant value has been accessed with the subscript operator and the subscript value used is outside the valid range for the constant. *value* is the dimension number that was subscripted.

symbol '*name*' multiply defined in symbol table

Named symbol has been multiply defined in the configuration source.

uninitialised symbol '*name*' in expression

Named symbol, which is of arithmetic type, has been used in an expression and has not been assigned any value.

unterminated character constant

A character constant has been specified where a closing quote has not been found before the end of the line.

unterminated comment statement

A comment has been started and has not been terminated before the end of the file.

unterminated string constant

A string constant has been specified where a closing double quote has not been found before the end of the line.

unused connector '*name*' in placement

Named connector has not been used in a connect statement and has been used in a place statement.

value for attribute '*name*' out of range

Named attribute has been assigned a value that is not in the valid range for the attribute, for example, a negative value for the `memory` attribute of a processor.

zero length character constant

A zero length character constant has been specified.

12.4.3 Serious messages

The following diagnostic messages are generated at severity level *Serious*.

TCOFF descriptor, illegal number of dimensions, *value*
TCOFF descriptor, illegal type for *name*, *type*
TCOFF descriptor, missing (, found *char*
TCOFF descriptor, missing), found *char*
TCOFF descriptor, missing :, found *char*
TCOFF descriptor, missing ? or !, found *char*
TCOFF descriptor, missing OF for CHAN or PORT
TCOFF descriptor, missing], found *char*
TCOFF descriptor, missing occam PROC keyword
TCOFF descriptor, missing occam identifier
TCOFF descriptor, overflow in dimension size

TCOFF descriptor, undefined channel parameter
TCOFF descriptor, unknown occam parameter type
TCOFF descriptor, unknown occam process type
TCOFF format, define main undefined
TCOFF format, descriptor and define main mismatch
TCOFF format, descriptor undefined
TCOFF format, expected library file
TCOFF format, illegal code size, *value*
TCOFF format, illegal entry offset, *value*
TCOFF format, illegal format origin symbol, *string*
TCOFF format, illegal scalar size, *value*
TCOFF format, illegal vector size, *value*
TCOFF format, module not found in library file
TCOFF format, multiple code sections
TCOFF format, multiple define mains
TCOFF format, multiple descriptors
TCOFF format, multiple origin symbols
TCOFF format, multiple virtual sections
TCOFF format, unexpected error mode, found none
TCOFF format, unexpected language, found *value*
TCOFF format, unexpected library file
TCOFF format, unexpected library tag, found *value*
TCOFF format, unexpected tag, found *value*
TCOFF format, unexpected transputer type, found none

These messages indicate an error in the format of the object file specified to the configurer in a use statement.

12.5 icconf error messages

This section documents command line and system errors (other than configuration source diagnostics) generated by `icconf`. Such errors are generated at severities *Serious* and *Fatal*.

The display format for error messages is described in section A.6.1.

12.5.1 Serious errors

The following errors are generated at severity level *Serious*.

ROM memory size undefined

The **RA** or **RO** options have been used and no **RS** option has been specified.

illegal ROM memory size, *value*

Value specified for the RS option is not greater than zero.

illegal format ROM memory size, *string*

An illegal format memory size value has been specified for the RS option. *string* is the illegal format memory size.

internal token buffer overflow, *value*

An internal buffer used for storing the current input line has overflowed causing the error. *value* is the size of the internal buffer in bytes.

multiple ROM memory sizes, *string*

The RS option has been specified more than once. *string* is the latest value for the RS option.

multiple input file names, *string*

An input file name has been specified more than once. *string* is the latest input file name.

multiple output file names, *string*

The O option has been specified more than once. *string* is the latest value for the O option.

multiple processor names, *string*

The P option has been specified more than once. *string* is the latest value for the P option.

options G and RA or RO are incompatible

The G and RA or RO options have been specified.

options XM and XO are incompatible

The XM and XO options have been specified.

processor name undefined

The RA or RO options have been used and no P option has been specified.

too many errors occurring, *value*

Number of errors exceeds maximum number allowed. *value* is the maximum number of errors allowed.

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

An attempt to open a file failed due to either the file not existing or an error occurring in the host file system. *value* is the error number for the failure.

unexpected command line token, *string*

A token has been specified on the command line to the configurer that is not recognised as a valid option string.

12.5.2 Fatal errors

The following errors are generated at severity *Fatal*.

illegal string length (*value***)**

A string length has been input from an object file which exceeds the maximum string length for an object file. *value* is the illegal string length found.

unable to allocate heap memory

Amount of memory available to the configurer is insufficient for configuring the configuration source.

unable to close (*value***)**
unable to read (*value***)**
unable to seek (*value***)**
unable to tell (*value***)**
unable to write (*value***)**

These messages are generated as a result of an error occurring in the host file system. *value* is the error number for the failure.

unexpected end of input

The end of the input has been found unexpectedly in an object file.

13 icollect – code collector

This chapter describes the code collector tool **icollect** which generates executable files for single and multitransputer programs, from configuration data files and linked units respectively. The tool is also used to create files for input to the EPROM programmer tool **ieprom**, and to generate files that can be dynamically loaded by application source code.

13.1 Introduction

icollect generates bootable files for transputer programs and other executable files in special formats. Bootable files are transputer executable files containing distribution and bootstrap information which can be directly loaded onto the hardware down a transputer link. The command line default is to generate a bootable file for a networked program from a configuration binary file; single processor operation and special outputs are selected by specific command line options.

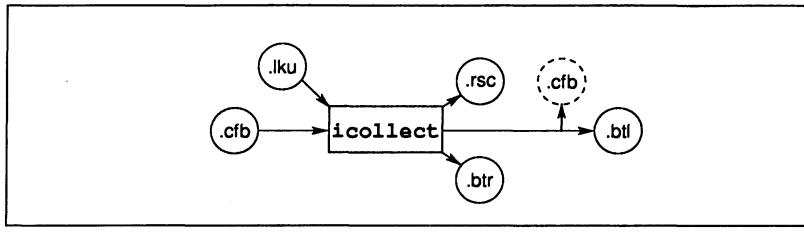
The bootable file contains all the information for loading and running the program on a specific network of processors. The file includes data that controls the distribution of code on the network and self-booting code for each processor. Bootable programs are self-distributing and self-starting and can be loaded directly onto the transputer hardware using **iserver**.

For multitransputer programs the input file is a configuration data file (by default, a file with the **.cfb** extension) created by the configurer from a configuration description. The file describes the placement of processes and channels on the processor network in a special format which can be read by the collector.

For single transputer programs the input file is a single linked unit (by default, a file with the **.lku** extension), to which bootstrap and system code is added for a single processor.

icollect can be directed to generate output files in a special format for processing by the **ieprom** tool, and executable code with no bootstrap or system process information, intended for dynamic loading by a high level language program.

The main inputs and outputs of the collector tool for bootable programs are shown below.



13.2 Running the code collector

The code collector is invoked using the following command line:

▶ **icollect** *filename* {*options*}

where: *filename* is a configuration data file created by **icconf** or a single linked unit created by **ilink**.

options is a list of options from the following tables.

> Options must be preceded by '-' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
B *filename*	Uses a user-defined bootstrap loader program in place of the standard bootstrap. The program is specified by *filename* and must conform to the rules described in appendix D. This option can only be used with the 'T' option (single processor mode) and cannot be used with the 'RA' and 'RO' options.
C *filename*	Specifies a name for the debug data file. A filename must be supplied and is used as given. Only valid when accompanied by the 'T' option and invalid if used with the 'D' or 'K' options.
D	Disables the generation of the debug data file for single transputer programs. Can only be used with the 'T' option.
E	Changes the setting of the Halt On Error flag. HALT mode programs are converted to not stop when the error flag is set, and non-HALT mode programs to stop when the error flag is set. Can only be used with the 'T' option.
I	Displays progress information as the collector runs.
K	Creates a single transputer file with no bootstrap code. Can only be used with the 'T' option. If no file is specified the output file is named after the input filename and given the `.rsc` extension.
L	Loads the tool onto the transputer board and terminates.
M *memorysize*	Specifies the memory size available (in bytes) on the root processor for single transputer programs. Can only be used with the 'T' option. *memorysize* can be specified in Kilobytes and Megabytes using the 'K' or 'M' suffixes. *memorysize* may also be specified in hexadecimal using the '#' or '$' prefixes.
O *filename*	Specifies the output file. A filename must be supplied and is used as given.
RA	Creates a file for processing by `ieprom` into a boot from ROM file to run in RAM. If no output file is specified the file is given the `.btr` extension. If the input is a configuration binary file it must have been created using the `iconf` 'RA' option.
RO	Creates a file for processing by `ieprom` into a boot from ROM file to run in ROM. If no output file is specified the file is given the `.btr` extension. If the input is a configuration binary file it must have been created using the `iconf` 'RO' option.

Option	Description
RS *romsize*	Specifies the size of ROM on the root processor. Only valid when used with the 'RA' or 'RO' options. *romsize* must be given in decimal format and can be followed by 'K' or 'M' to indicate Kilobytes or Megabytes. *romsize* must match the romsize specified to `icconf` if used.
S *stacksize*	Specifies the extra runtime stack size for single transputer programs. Can only be used with the 'T' option. *stacksize* must be given in decimal format.
T	Creates a bootable file for a single transputer. The input file specified on the command line must be a linked unit. This option cannot be used for programs linked with the *reduced* runtime library.
XM	Directs the transputer-hosted version of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted version of the tool to be executed once on the transputer board and then terminate.

13.2.1 Examples of use

Example A (single processor mode):

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icollect hello.lku -t
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icollect hello.lku /t
iserver /sb hello.btl /se
```

Example B (configured program mode):

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icconf hello.cfs
icollect hello.cfb
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icconf hello.cfs
icollect hello.cfb
iserver /sb hello.btl /se
```

13.2.2 Input files

The input file is either a configuration data file generated by **iconf**, or a linked unit generated by **ilink**. By default the collector assumes a configuration data file; for linked units that are to be processed for single transputers the '**T**' option must be specified. Incorrect format input files generate an error message and no output is produced.

13.2.3 Output files

The main output file is a binary file that can be loaded directly onto the transputer hardware down a transputer link, whether for a single transputer or a multitransputer network. This type of file is known as a *boot from link* program. If no filename is specified the output file is named after the input file and given a **.btl** extension. If an output filename is specified the file is given the specified name.

Files created using the '**RA**', '**RO**', and '**K**' options are given special extensions (if no output filename is specified) which indicate the file type. File types created for each of the options are listed below.

Option	File created
K	.rsc
RA	.btr
RO	.btr

13.2.4 Non-bootable files

Files created with the '**K**' option are non-bootable files which can be dynamically loaded by a program or manipulated at runtime.

Non-bootable files consist essentially of program code preceded by a number of words of runtime data. The sequence of data and code blocks in the file is summarised in the following table. Descriptions of the data items immediately relating to the program block are given after the table.

No of words (long ints)	Data	Unit
one	Interface descriptor size	bytes
Set by above	Interface descriptor	–
one	Compiler id size	bytes
Set by above	Compiler id	–
one	Target processor type	–
one	Version number	–
one	Program scalar workspace requirement	words
one	Program vector workspace requirement	words
one	Static size	words
one	Program entry point offset	bytes
one	Program code size	bytes
Set by above	Program code block	–

Target The processor type or transputer class for which the program was compiled.

Version The format version number of the file. This can be 10 or 11 in the TCOFF system. For programs compiled with `icc` it will always be 11, which indicates the presence of a static data parameter. A value of 10 indicates no static parameter and is used to identify code written using other INMOS language toolsets.

Scalar workspace Specifies the size of the workspace required for the linked program's runtime stack.

Vector workspace Specifies the size of the workspace required for the linked program's vector (array) data.

Static size Specifies the size of the static area.

Entry point offset Indicates the offset in bytes of the program entry point from the base of the code block.

Code size Indicates the size of the program code in bytes.

Code The program code.

13.2.5 Boot-from-ROM options

The boot-from-ROM options '`RA`' and '`RO`' produce code that can be loaded into EPROM using the `ieprom` tool. Both options apply only to code running on the

root transputer of a network; processors on the network connected to the root transputer are booted from the root transputer links.

'RA' generates code which is executed from RAM. The code is copied from ROM into RAM at runtime. 'RO' generates code which is directly executed from ROM.

RAM executable code can be used for applications which are to be executed from fast RAM, and for code which may be user-modified. ROM executable code requires no external RAM and can be used to create a truly embedded system.

Configured programs for loading into ROM must have been created using the same configurer option ('RO' or 'RA' as appropriate) that is supplied to the collector.

13.2.6 Debug data file

For single transputer programs only, the collector automatically generates a configuration binary file for reading by the debugger. By default the filename stem is taken from the output file and the '.cfb' extension is added. If the 'C' option is specified the filename is used as supplied. Generation of the debug data file can be disabled by specifying the 'D' option.

13.2.7 Alternative bootstrap loaders

If not otherwise specified, `icollect` uses the standard INMOS primary bootstrap loader. The correct code for the application program is chosen from a library of bootstraps compiled for different transputer types and error modes.

The collector can be directed to use other bootstrap loader programs defining different loading sequences by specifying the 'B' option. The option directs the collector to append a user-defined loader program in place of the standard bootstrap loading sequence.

User-defined bootstraps must be created according to certain rules, illustrated by the standard INMOS bootstrap which is listed in appendix D along with the standard Network Loader. The listing is fully commented and can be used as a template to design and code your own bootstrap sequence.

13.2.8 Small values of IBOARDSIZE

If **IBOARDSIZE** is set to a small value, for example if the value specified is invalid and it is set to 0 by default, the collector generates a warning message. Very small values of **IBOARDSIZE** (including zero) are detected at runtime and

prevent the program from being run.

13.3 Error messages

This section lists error messages generated by `icollect`. The messages are listed under severity headings in alphabetical order, omitting the introductory information (error severity and filename data).

`icollect` generates errors of severities *Warning* and *Serious*. Serious error cause the tool to terminate without producing any output.

13.3.1 Warnings

The following messages are prefixed with '`Warning-`'. They are only generated when the '`T`' option is used (single processor mode).

Flip error mode ignored with user bootstrap

The '`E`' option is ignored when a user-defined bootstrap is specified since the collector will only accept a single linked unit as a bootstrap.

Strange board size for sixteen bit processor : Setting to zero

The memory size specified exceeds the addressing capacity of a 16 bit processor (64 Kbytes). The collector uses a memory size of zero for the rest of the build.

13.3.2 Serious errors

The following errors are prefixed with '`Serious-`'.

Address space for target processor exhausted

The address space required by the program is greater than 64Kbytes, the maximum addressable space on a 16-bit processor.

Bootstrap file already specified

More than one bootstrap file was specified. Only one file is allowed.

Bootstrap filename too long

The maximum length allowed for the bootstrap filename is 255 characters.

Bootstrap is greater than 255 byte in library file

The library bootstrap is too large. This should only occur if the library file is invalid or corrupt.

Cannot have both rom types

'RA' and 'RO' options are mutually exclusive and cannot both be specified on the same command line.

Cannot have configured and memory size

The memory size option is incompatible with building a bootable program for a configuration binary file.

Cannot have configured and non bootable file

The collector cannot generate both a network loadable file and a non-bootable file simultaneously for the same program.

Cannot have rom and non bootable file

The collector cannot generate both a ROM-loadable file and a non-bootable file simultaneously for the same program.

Cannot open file *filename*

Host file system error. The file specified cannot be opened.

Command line parsing error at *string*

Unrecognised command line option.

Debug file already specified

More than one debug was file specified. Specify one only.

Dynamic memory allocation failure

Memory allocation error. The collector cannot allocate the required amount of memory for its internal data structures.

Error in writing to debug file

Host file system error. The debug file could not be written. This message will only appear if the collector is invoked with the 'T' option (single

processor mode).

Expected end tag found not present in .cfb file

A specific end tag is missing in the configuration binary file. Either the file is corrupted or the versions of **icollect** and **icconf** used are incompatible.

Illegal tag found in .cfb file

Incorrect format configuration binary file, recognised as an illegal tag. Either the file is corrupted or the versions of **icollect** and **icconf** used are incompatible.

Illegal language type found in input file

Source language used to create the file is not supported by the collector. Less likely, but possible, is that the file was created using an incompatible (possibly earlier) version of a tool.

Illegal process type

Unrecognised process type. Either the file has been corrupted or the versions of **icollect** and **icconf** used are incompatible.

Illegal processor type

Unrecognised processor type. Either the file has been corrupted or **icollect** and **icconf** are incompatible.

Illegal tag found in input file : *filename*

Incorrect format input file. The most likely reason for this error is that an incorrect file has been specified. Other less likely but possible explanations are that the file was created using an earlier or incompatible version of one of the tools, or that the file has become corrupted.

Input file already specified

More than one input file specified on the command line.

Input file has not been linked *filename*

The collector accepts only linked files, either directly when using single processor operation, or indirectly via entries in the configuration binary file. This message can be generated if the file was created using a

previous version of a tool, or if the file is corrupt.

Input file is of incorrect type : *filename*

If the '**T**' option is specified (single processor program) the input file must be a single linked unit (**.lku** type). If the '**T**' option has not been specified the input file must be a configuration binary file (**.cfb** type).

Input filename too long

The maximum length allowed for the input filename is 256 characters.

Linked unit file in cfb and linked unit in input file found do not match : *filename*

The linked file specified in the configuration binary and the one found the collector do not match.

Linked unit module not found in : *filename*

The required library module is missing or has been corrupted. This message is generated when an incorrect version of the library is installed.

Memory size already specified

Memory size must be specified once only.

Memory size string invalid

Memory size must be given in decimal or hex. Hex numbers must be introduced by '#' or '$'.

Memory size string too long

Specified memory size is too large.

More than one parameter statements

The collector expects only one *parameter* statement per processor. Either the file has been corrupted or the versions of **icollect** and **icconf** used are incompatible.

No debug and debug output file specified in command line

Options '**D**' (disable debug) and '**C**' (debug filename) cannot be used together.

No input file specified

One, and only one, input file must be specified on the command line.

No parameter descriptor present in input file : *filename*

The formal parameter descriptor in the input file is not present. This usually means that the process has not been linked with a main entry routine. This message will only appear if the collector is invoked with the '**T**' option (single processor mode).

Output file already specified

More than one output file was specified. Specify only one.

Output filename too long

The maximum length allowed for the output filename is 256 characters.

Parameter descriptor error in input file : *filename*

The formal parameter descriptor in the input file is not of the correct form, indicating that the process interface is not one recognised by the collector. This message will only appear if the collector is invoked with the '**T**' option (single processor mode).

Program configured for boot from ROM command line is boot from link

The specified configuration binary file was created for either ROM or RAM, and neither has been specified to `icollect`.

Program configured for running in RA mode command line is RO mode

Wrong mode specified, or incorrect option given to `icconf` when the specified configuration binary file was created. RO and RA modes are mutually exclusive.

Program configured for running in RO mode command line is RA mode

Wrong mode specified, or incorrect option given to `icconf` when the specified configuration binary file was created. RA and RO modes are mutually exclusive.

Rom size already specified

ROM size must be specified once only.

Rom size in input file and command line do not match

The ROM size specified on the command line does not match that specified to `iconf` when the input file was created.

Rom size not specified

A ROM size must be specified because the input file is configured for loading into ROM.

Rom size string invalid

ROM size must be given in decimal.

Rom size string too long

ROM size specified was too large.

Stack size already specified

Stack size must be specified once only.

Stack size string invalid

Stack size must be specified in decimal format.

Stack size string too long

Specified stack size was too large.

Strange function or attribute for linked unit in : *filename*

The collector has found an unfamiliar value in the input file. Either an old version of a tool was used in the creation of the input file, or the input file has been corrupted.

System error

Host system error has occurred, probably when accessing a file. This message may be generated when a file is read and its contents seem to have changed.

Unexpected end of file : *filename*

One of the files specified in the configuration binary has ended prematurely. *filename* identifies the offending file. If the message 'Suspect

corrupted file' is substituted for *filename* then the file is corrupted.

User bootstrap not allowed when program is configured

User defined bootstrap loaders can only be used with single processor programs.

User bootstrap not allowed with rom option

User defined bootstrap loaders cannot be used with ROM-loadable code.

User bootstrap type does not match that of linked unit

Either the target processor type or the error mode of the bootstrap code does not match that of the input file.

14 icvlink – file format convertor

This chapter describes the file format convertor tool **icvlink** which converts object files from *L*inker *F*ile *F*ormat (LFF) to *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF). The chapter begins with a short introduction to the tool and then describes how it is used. The chapter ends with a list of error messages which may be generated by **icvlink**.

14.1 Introduction

Earlier compilers and INMOS toolsets targetted at the transputer produced object files in LFF. Examples of such products are the 3L and INMOS Parallel C, occam, FORTRAN and Pascal compilers.

All object files produced by the latest INMOS Toolsets are generated in a format known as *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF). Input files for the linker, librarian, and lister tools, supplied with these toolsets, *must* be in TCOFF.

icvlink enables code compiled in LFF to be used with later versions of the tools without needing to recompile. In particular it enables existing software to make use of the new configuration language implemented by the current toolset.

The conversion to TCOFF may take place at different stages in the development process depending on the user's requirements. Figures 14.1 to 14.3 illustrate three different approaches to using **icvlink**. Notice that in all three approaches the conversion is performed before the configuration stage.

In figure 14.1, compiled object and library modules are processed by the convertor and then linked using the current toolset linker **ilink**. Converted library modules have to be processed by the current toolset librarian **ilibr** in order to create TCOFF library modules, see section 14.2.2.

Figure 14.2 illustrates how existing compilation and library modules may be linked using a previous version of the linker to produce a linked object file in LFF. This file may then be converted to TCOFF and the current toolset linker **ilink** used to create a linked object file in TCOFF.

Figure 14.3 illustrates an extension to the second approach, where the TCOFF file produced by the conversion is linked with modules compiled by the current toolset compiler. The linking process is performed by employing the same method of linking that is used for mixed languages.

The shaded symbols, in the figures, represent both i/o files in LFF format and previous issues of particular tools. **Note**: where *txx* has been used it would be equally valid to use **.bin** (see section 14.2 below).

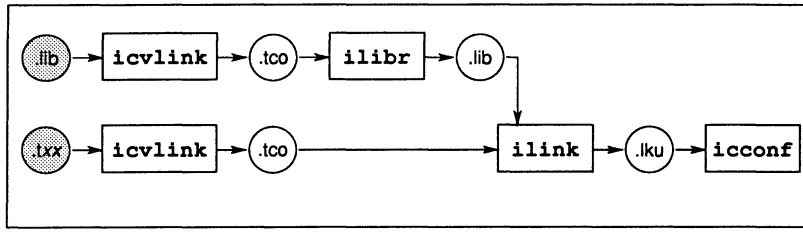


Figure 14.1 Converting compilation and library modules

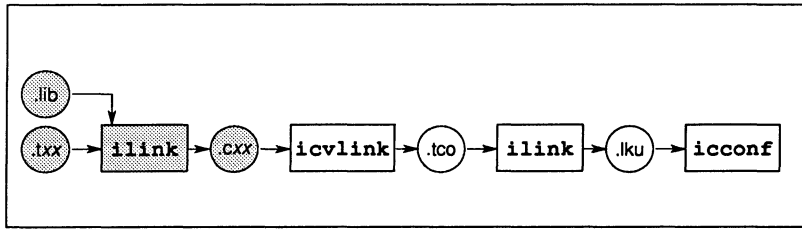


Figure 14.2 Converting linked object module

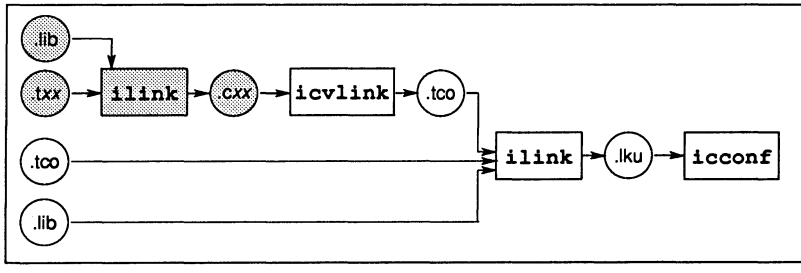


Figure 14.3 Conversion followed by linking for mixed languages

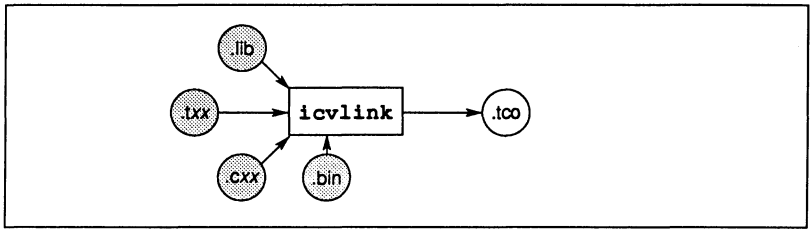
When source code is available it is recommended that the source code is recompiled using the compiler supplied with this toolset rather than using **icvlink**. If, however, the source code is not available or recompilation is likely to be difficult, then **icvlink** should be used, following one of the approaches outlined above.

Programs which have been converted should in general be kept separate from programs developed with the current toolset. This is because of differences in the supplied libraries and in the implementation of the different versions of the compilers and toolsets. If it is necessary to combine old and new software,

the modules should be linked using the methods described for mixed language programming, see chapter 9.

14.2 Running the format convertor

The format convertor operates on a single input file. This file may be a single module or a library. The operation of the format convertor in terms of standard extensions is shown below.



Note: The file extensions of the input files, pertain to default file extensions used by previous issues of INMOS toolsets, where:

`.lib` is the extension of a library file.

`.t`*xx* is the extension of a compiled occam file.

`.c`*xx* is the extension of a linked unit.

`.bin` is the extension of a compiled C or FORTRAN file.

To invoke the file format convertor use the following command line:

▶ **`icvlink`** *filename* {*options*}

where: *filename* is the name of the file to be converted. Any string not recognised as an option is treated as an input filename.

options is a list of options given in table 14.1.

Options must be preceded by '-' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
Options may be entered in upper or lower case and can be given in any order.
Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
D	Forces a TA module to be converted into both a new TA module and a T8 module. Forces a TC module to be converted into both a T5 and a T8 module. This option is only for use with library modules.
I	Displays progress information as the conversion proceeds.
L	Loads the tool onto a transputer board and then terminates.
O *filename*	Specifies an output file. If no output file is specified the name is taken from the input module and a .tco extension is added. If more than one output file is specified then the last one takes precedence.
P	Forces TA and TC modules to be converted to T8 modules.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

Table 14.1 icvlink command line options

Examples

```
icvlink myprogc.bin
```

In this example icvlink is used to convert an object file, produced by the INMOS 3L Parallel C compiler. The output filename will default to myprogc.tco.

```
icvlink myprog.t4x
```

In this example icvlink is used to convert an occam object file which has been compiled for a T4 series transputer. The output filename will default to myprog.tco.

14.2.1 Default command line

A set of default command line options can be defined for the tool using the `ICVLINKARG` environmental variable. Options must be specified using the syntax required by the command line.

14.2.2 Input files

The format convertor will accept a compiled object file, a linked object file or a library file, in LFF format, as input. The following sections describe the use of the format convertor in the context of these file types.

Compiled object files

The format convertor may be used to convert any compiled object files. The convertor will produce compiled modules in TCOFF format. Any libraries required to be linked with the compilation modules must also be converted (see below), before the linker `ilink` can be used to produce the linked object file.

Library files

The format convertor will convert a library file which is in LFF format to the new TCOFF format but it will not generate a new library file. When a library is converted the resulting file contains a concatenation of all the converted modules. In order to create a library file the librarian tool `ilibr`, supplied with this toolset, must be used to prepend the library index.

Linked object files

Linked object files in LFF format may also be converted into TCOFF format.

The procedure for converting linked files is similar to that for converting compiled object files. The format convertor will convert a linked object file which is in LFF format into a TCOFF format file. This file may then be supplied as an input file to the linker tool `ilink` in order to produce a linked object file in the new format.

14.2.3 Output files

The format convertor creates a single TCOFF object module. As indicated above, if either a library or linked object module is used as input then the output module must be processed by the current `ilibr` or `ilink` tools.

14.3 Transputer classes and error modes

Both the members and the meaning of the different transputer classes has changed for this issue of the toolset. **icvlink** therefore has to impose a transputer class on any module whose class has no direct representation in the current toolset. This also applies to error modes. The following rules are used for transputer classes and error modes:

- The error mode UNDEFINED is converted to UNIVERSAL.
- Transputer class TA does not change name but note that the meaning of this class has changed.
- Transputer class TC is converted to transputer class T5.

For more information on transputer classes see section 5.3.

The command line options '**D**' and '**P**' can be used to override these rules. The command line option '**P**' causes TA and TC modules to be converted to T8 modules. The '**D**' option is designed to be used when converting libraries that contain TA and TC modules. When a TA library module is converted with this option two modules will be generated by the conversion; one '*new style*' TA module and one T8 module. For a TC library module converted with the '**D**' option, a T5 and T8 module will be created.

The '**P**' option may be used to convert any compiled, library or linked object modules. The '**D**' option, however, is restricted to converting library modules, because the linker can selectively load library modules whereas it cannot selectively load compilation modules.

14.4 Summary of rules for using **icvlink**

1 When source code is available **icvlink** should not be used. Instead the source code should be recompiled using the compiler supplied with this toolset.

2 The libraries supplied with this toolset must not be linked with converted object modules. Instead the library files originally called by the converted modules must also be converted so that the modules may be linked correctly.

3 If converted modules are to be used in conjunction with modules compiled by the current toolset, then they must be linked by using mixed programming techniques. In general converted object and library modules should be used in isolation of any new development.

14.5 Error messages

This section lists each error and warning message that can be generated by the convertor. Messages are in the standard toolset format which is explained in appendix A.

14.5.1 Serious errors

filename **- bad format:** *reason*

The named file does not conform to a recognised INMOS file format or has been corrupted.

Could not open for input

The named file could not be opened for reading.

Could not open for output

The named file could not be opened for writing.

No input file supplied

No file name has been placed on the command line.

Only one input file allowed

More than one file name has been placed on the command line.

Parsing command line *token*

An unrecognised token was found on the command line.

Promote and duplicate options conflict

The `P` (promote) and `D` (duplicate) options have conflicting meanings and should not be used in conjunction.

15 idebug – debugger

This chapter is describes the network debugger tool **idebug**. It begins by describing the command line syntax and shows how to invoke the debugger in the two main debugging modes. The rest of the chapter lists and describes in detail the symbolic debugging functions and Monitor page commands and ends with a list of error messages.

15.1 Introduction

The network debugger **idebug** is a special purpose debugger for transputers. It can be used to examine stopped programs (post-mortem debugging) or to execute programs interactively (breakpoint debugging).

Programs can be analysed using the *symbolic* functions which operate using source code symbols or the *Monitor page* commands which operate at memory and processor level. Symbolic and Monitor page environments are separate but can be recalled from each other at will.

Symbolic functions allows files to be examined, variables inspected, and procedures traced, from source code level. Monitor page commands allow transputer memory to be examined and processor state to be determined anywhere on the network. Symbolic and Monitor page environments can be recalled from each other at any time.

15.1.1 Post-mortem debugging

Post-mortem mode debugging allows stopped programs to be analysed from the residual contents of transputer memory or from a network dump file. Programs that run on the root transputer must be debugged from a memory dump file because the debugger overwrites the root transputer's memory. The memory dump file is created using the **idump** tool.

15.1.2 Breakpoint debugging

Breakpoint mode debugging allows transputer programs to be executed interactively using breakpoints set in the code. Breakpoints can be set symbolically on lines of source text or at transputer memory addresses, and values can be modified in transputer memory to show the effect of changing variables.

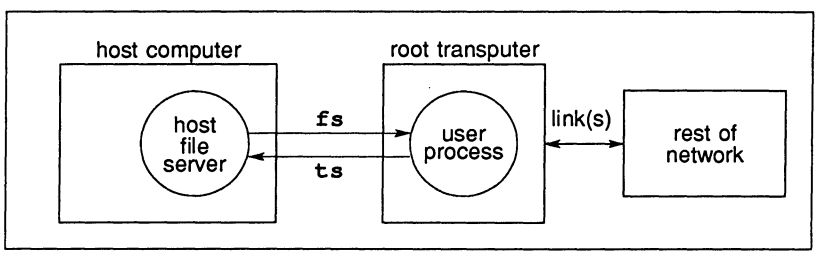
Certain symbolic functions and Monitor page commands are only available in breakpoint mode.

15.2 The root transputer

idebug can be used to debug single and multitransputer programs. The techniques and commands to use when invoking the debugger differ slightly according to whether or not the program (or a process forming part of the program) runs on the root transputer, and according to the debugging mode (post-mortem or breakpoint).

The **root** transputer is the name given to the processor that is directly connected to the host computer. In a transputer network that is connected to the host it forms the root of the network. The debugger always runs on the root transputer, which must be a 32-bit transputer with at least one Megabyte of memory.

The relationship of the root transputer to the host computer and the rest of the network is illustrated below.



Two procedures are used to debug programs in post-mortem mode, depending on whether the application is configured to use the root transputer. Programs that use the root transputer are referred to in this chapter as **R-mode** programs, and programs that do not use the root transputer are referred to as **T-mode** programs. Command line options are used to select the correct mode of operation for **idebug**.

To avoid the need for a memory dump applications configured to use the root transputer can be skip loaded. Skip loading requires at least one extra processor on the network but speeds up debugging considerably and is the recommended method where more than one processor is available. **iskip** can be used to skip any number of processors on a network by invoking the tool successively.

15.2.1 Board wiring

Before any program can be debugged in post-mortem mode on a transputer board the Analyse signal must be asserted on the network once, and once only. Because different procedures must be adopted for programs which do and do not use the root transputer, the debugger cannot assert the signal automatically and it must be asserted by passing the appropriate **iserver** option from the **idebug**

command line. Table 15.2 gives a summary of the command sequences to use for the two program modes on different board types.

15.2.2 Post-mortem debugging R-mode programs

Code running on the root transputer and loaded with **iserver** directly is debugged in post-mortem mode from a *memory dump* file which is specified by the '**R**' option. The memory dump file must be created using the **idump** tool before the debugger is invoked. Code on other transputers is debugged down transputer links in the normal way.

In R-mode programs **idump** asserts the Analyse signal and the '**SA**' option is not required on the **idebug** command line. In fact a second assertion of the signal would cause data in the memory to become corrupted. If **idump** is not invoked then the debugger cannot load onto the root transputer and a booting error is reported.

Details of the **idump** and **iskip** tools can be found in chapters 16 and 25 respectively.

15.2.3 Post-mortem debugging T-mode programs

T-mode programs are loaded using **iskip** and subsequently debugged using the '**T**' option to specify the root transputer link to which the network is connected. The '**SA**' server option must also be added to the **idebug** command line in order to assert **Analyse**.

If the '**SA**' option is not given, the debugger is not booted onto the root transputer and the server aborts with an error message. If the server is inputting data at the time some corruption of the data may occur. The debugger should then be reinvoked with the correct options.

15.2.4 Post-mortem debugging from a network dump file

To suspend a post-mortem R or T debugging session without losing the original context, the Monitor page '**N**' command can be used to dump the entire state of a network into a network dump file (including Freespace if required). The debugger can then be invoked on the file without being connected to the network.

Notes: This option will only work for programs that have not been interactively breakpoint debugged.

Memory dump files and network dump files are not the same: the former con-

tains a single processor's memory image while the later contains data about a complete network. They are also in different formats.

15.2.5 Debugging a dummy network

The debugger may be used to debug a program using dummy data. Using the debugger command line 'D' option which simulates the contents of memory locations and registers, static features of a program may be examined. This is useful to determine processor connectivity and memory mapping for each processor in the network. This option may also be used to explore the features of the debugger.

15.2.6 Methods for breakpoint debugging

Breakpoint mode debugging does not require use of the memory dump tool because the program is automatically skip loaded over the root transputer where the debugger is running. However, like all skip loads it requires an extra processor in the network.

15.3 Running the debugger

The debugger is invoked using the following command line:

▶ **idebug** *filename* {*options*}

where: *filename* is the program bootable file.

options is a list of one or more options from table 15.1.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
B *linknumber*	Interactive breakpoint debug a network that is connected to the root processor via link *linknumber*. **idebug** executes on the root processor.
	Must be accompanied by the iserver '**SR**' option.
M *linknumber*	Postmortem debug a previous interactive debugging session. **idebug** executes on the root processor.
	Must be accompanied by the iserver '**SA**' option.
T *linknumber*	Postmortem debug a program that does not use the root processor, on a network that is connected to link *linknumber*. **idebug** executes on the root processor.
	Must be accompanied by the iserver '**SA**' option.
R *filename*	Postmortem debug a program that uses the root transputer. *filename* is the file that contains the contents of the root processor (created by **idump**). The file is assumed to have the extension **.dmp** if none is supplied.
N *filename*	Postmortem debug a network from a network dump file filename (created by **idebug**). The file is assumed to have the extension **.dmp** if none is supplied.
	Must be accompanied by the iserver '**SR**' option.
C *type*	Specify a processor type (e.g. T425) instead of a class (e.g. TA) for programs that have not been configured.
D	Dummy debugging session. Can be used for familiarisation with the debugger or establishing memory mappings.
	Must be accompanied by the iserver '**SR**' option.
A	Assert subsystem analyse. Directs the debugger to assert **Analyse** on the network connected to the root processor.
S	Ignore subsystem error status when breakpoint debugging.
I	Display debugger version string.
	Must be accompanied by the iserver '**SR**' option.

Table 15.1 Debugger command line options

15.3.1 Environment variables

idebug requires three environment variables to be set up on the host system:

ITERM	Defines key mappings for debugger symbolic functions and some Monitor page commands.
IDEBUGSIZE	Defines the amount of memory available on the root transputer board. This variable must be specified for **idebug** to work correctly (**idebug** requires at least 400Kbytes of available root transputer memory).
IBOARDSIZE	The amount of memory available for the application program. Required for single transputer programs (created from linked units using **icollect** with the 'T' option and without the 'M' option), where the memory size was not specified.

Details of how to set up the variables can be found in the Delivery Manual that accompanies the release.

15.3.2 Program termination

If the program terminates on issuance of the server terminate command by the C runtime system the following message is displayed:

```
[Program has finished - hit any key for monitor]
```

The debugger can be re-entered after server termination by pressing any key. The final state of the network can be examined using the full range of Monitor page and symbolic commands.

The exit status returned by the program is displayed on the Monitor page.

If the program contains independent processes which require no communication with the server the debugger allows the program to be resumed. In this case the debugger displays the following warning message:

```
[Warning: The server has been terminated by the program]
```

15.3.3 Post-mortem mode invocation

To invoke the post-mortem debugger use the appropriate command from the following list.

Command lines are duplicated in UNIX and MS-DOS/VMS formats. Use the appropriate command line format for your system.

Note: Commands are given for a B008 board wired *subs*. For the commands to use on other board types see section 15.4.

```
idebug bootablefile -t linknumber -sa
idebug bootablefile /t linknumber /sa

idebug bootablefile -r filename
idebug bootablefile /r filename

idebug bootablefile -n filename -sr
idebug bootablefile /n filename /sr

idebug bootablefile -m filename -sa
idebug bootablefile /m filename /sa
```

where: *bootablefile* is the program bootable file.

linknumber is the number of the link of the root processor which is connected to the network.

filename is a network dump file or a root transputer memory dump file.

Use the '**t**' option for programs that do not use the root transputer, that is, those loaded by using **iskip**. The program is debugged from the program image that is resident in the memory of each transputer; the information about the rest of the network is extracted down the root transputer link. The '**t**' option produces faster debugging option because the root transputer memory image is not saved. However, the option does require an extra transputer on the network. The '**t**' option should be accompanied by the iserver '**sa**' option to assert Analyse on the network.

Use the '**r**' option for programs that use the root transputer in a network. The dump file is created by using **idump**, which produces a dump of the program image on the root transputer only; the debugger extracts information about other transputers on the network (if applicable) via the root transputer links.

Use the '**n**' option to debug programs without access to the original network of transputers. This is effectively debugging off-line. The network dump file is generated by the **idebug** Monitor page '**N**' command (only for programs that have not been breakpoint debugged). The '**t**' option should be accompanied by the iserver '**sr**' option to reset the network.

Use the '**m**' option to debug a previous breakpoint debugging session where either the network has crashed (error flag was set) or you have used the host BREAK key to terminate the debugger. This option is the same as the '**t**' option but informs the debugger the breakpoint runtime kernel is present. The

'**m**' option should be accompanied by the iserver '**sa**' option to assert Analyse on the network.

Symbolic functions and Monitor page commands that support breakpointing are absent in the post-mortem debugger.

Reinvoking the debugger on single transputer programs

For programs running on a single transputer only and debugged from a memory dump file the debugger can be reinvoked on the same dump file by passing the '**SR**' option to **iserver** from the **idebug** command line. This option is required to reset the transputer before loading the debugger program, which is normally simulated by **idump**.

15.3.4 Breakpoint mode invocation

To invoke the debugger in breakpoint mode use one of the commands below.

Note: Commands are given for a B008 board wired *subs*. For the commands to use on other board types see section 15.4.

idebug *bootablefile* **-b** *linknumber* **-sr**
idebug *bootablefile* **/b** *linknumber* **/sr**

where: *filename* is the program executable file

linknumber is the number of root transputer link where the application network is connected.

In breakpoint mode **idebug** loads the bootable file directly onto the network and sets up a runtime kernel and virtual link system on each processor used by the program. **iserver** is not required to load the program, but an extra processor is required to run the debugger; the program is in effect 'skip' loaded.

Clearing error flags on transputer boards

Processors in the network with their error flags set can cause **idebug** to signal a crashed program even when they are not being used by the program. This is because **idebug** uses subsystem services to monitor error flag status throughout the network. A reliable method of clearing all of error flags on a network is to run a network check or worm program such as **ispy** before invoking **idebug**.

The **ispy** program is provided as part of the board support software for INMOS *i*q systems products. These products are available separately through your local

INMOS distributor.

An alternative method of ensuring that error flags are cleared on a network is to load a dummy process on each processor. The act of loading code onto the processor clears the error flag.

The following is an example of a dummy process which could be used to clear the error flag on a processor. The code simply starts up then shuts down immediately (**exit_terminate** (**exit_terminate** is used because the program is *configured*).

```
/*******************
*
*  Place this program on each processor
*  to clear the error flags.
*
*  Remember to use startup.lnk for the
*  root processor and startrd.lnk for
*  all other processors when linking.
*
*******************/


#include <misc.h>

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

{
exit_terminate (0);
}
```

Generate a linked unit containing the dummy process code for each processor on the network. Write a configuration description which places the linked units on each processor, collect the program, and load the resulting bootable file onto the network using **iserver**. The bootstrap code clears the error flag on each processor before loading the process code.

Note: When linking each process for subsequent configuration the process to be placed on the *root* processor must be linked with the full library; processes to be placed on other processors in the network can be linked with the reduced library.

Program loading

In breakpoint mode **idebug** loads the bootable program directly onto the network and sets up a debugging runtime kernel on each processor. **iserver** is not required to load programs for breakpoint debugging. An extra processor is required on the network to run a program in breakpoint mode because the program is in effect skip loaded.

When first invoked the breakpoint debugger immediately enters the Monitor page where the '**B**' (Breakpoint Menu) command can be used to set breakpoints before the program is started.

15.3.5 Function key mappings

All the debugger symbolic functions, and some Monitor page commands, are assigned to specific keys on the keyboard by the ITERM file (the file specified by the environment variable **ITERM**). For the correct keys to use on your terminal consult the keyboard layouts provided in the Delivery Manual that accompanies the release.

ITERM files are supplied with the release for terminals commonly used with your host system but may also be created to suit your own requirements. Details of the ITERM file and an example listing which illustrates the format can be found in appendix H.

Key-mapped symbolic functions and Monitor page commands are listed in section 15.6.2.

15.4 Debugging programs on different board types

On transputer boards the **Analyse** and **Reset** signals can be propagated from the root transputer in two ways, and this influences the options that must be used when debugging programs.

15.4.1 Subsystem wiring

On transputer boards the subsystem signal are either propagated unchanged to all transputers on the network (known as wired *down*), or the signals are connected to the subsystem port (wired *subs*) from where they are controlled by the board's root processor.

On B004 boards and on all boards where subsystem is wired in the same way **Analyse** must be asserted on the network before transputers can be accessed

by the debugger from the root processor. However, if **Analyse** is asserted more than once the program will be corrupted in transputer memory.

The wiring type can be identified by the hardware addresses of the three subsystem registers. B004-type boards use the following addresses:

Signal	Hardware address
Reset	#00000000
Analyse	#00000004
Error	#00000000

An example of a B004-type board is the IMS B404 TRAM. For details of the subsystem wiring on other boards consult the Datasheet or board specification.

In addition, TRAM boards and B004 boards differ in the way the subsystem port is used. On TRAMs the signals are propagated to all transputers on the network, whereas on B004 boards the signals are not propagated at all.

15.4.2 Debugging commands

The above conditions affect the commands you must use when debugging T-mode and R-mode programs. To simplify the selection of the correct command Table 15.2 has been constructed giving the command line options to use for different combinations of board type, subsystem wiring, and program mode.

Note: Command lines are given in the UNIX format ('-' option switch character) in order maintain simplicity in layout. For MS-DOS and VMS based systems replace '-' by '/' in all command lines.

For further details about loading programs see chapter 7.

15.4.3 Detecting the error flag in breakpoint mode

In breakpoint mode the debugger detects that a processor has its error flag set by use of the subsystem services. If your hardware is not wired up to use the subsystem services then the debugger is unable to detect when an error flag is set; this may cause the debugger to hang for no apparent reason. On such networks you should use the `iserver` '`SE`' option to detect when an error flag has been set. Note however that detection of an error flag set will terminate the debugger without warning.

Note: When using the debugger in breakpoint mode you should if possible wire your hardware up to use the subsystem services.

Board	Wiring	Mode	Command line(s) to use
TRAM	*down*	T	**idebug** *program* **-b** *linknumber* **-sr -se**† **-s**†
			idebug *program* **-m** *linknumber* **-sa**
			idebug *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
	subs	T	**idebug** *program* **-b** *linknumber* **-sr**
			idebug *program* **-m** *linknumber* **-sa**
			idebug *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
B004	*down*	T	**idebug** *program* **-b** *linknumber* **-sr -se**† **-s**†
			idebug *program* **-m** *linknumber* **-sa**
			idebug *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
	subs	T	**idebug** *program* **-b** *linknumber* **-a -sr**
			idebug *program* **-m** *linknumber* **-a -sa**
			idebug *program* **-t** *linknumber* **-a -sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename* **-a**

For MS-DOS and VMS based toolsets use the '**/**' option switch character.

Options on the **idebug** command line that are not debugger options are passed to **iserver**.

The '**si**' option may also be used on any command line to display activity information when loading the debugger.

Modes: R = program using the root transputer; T = program not using the root transputer, and debugged down a root transputer link.

† See section 15.4.3.

Table 15.2 Commands to use when debugging B004 and TRAM boards

15.5 Debugging programs on other boards

For hardware that does not adhere to the INMOS subsystem convention you will need to determine how the hardware is configured and the appropriate command

line options yourself.

You will probably need to use the **idebug** command line '**S**' option when breakpoint debugging in order to stop the debugger monitoring the subsystem error status, and the iserver '**SE**' option to determine when the error flag has been set.

15.6 Monitor page commands

This section lists and describes the Monitor page commands. The commands are tabulated in alphabetical order for easy reference. Where a command invokes an option submenu the operation of each option is described. Summaries of the commands can also be found in the Handbook that accompanies the ANSI C toolset release.

Command format

All Monitor page commands are either single letter commands or are invoked by a single function key press. Key mappings for the few general commands that use function keys can be found in the Delivery Manual that accompanies the release.

Specifying transputer addresses

Many Monitor page commands require a transputer address. If none is given the debugger assumes a default address when one is displayed with the prompt. The default address is the last address specified or located to and can be selected by pressing RETURN.

Addresses can be specified in decimal or hexadecimal format. Hexadecimal numbers must be given as a sequence of hexadecimal digits preceded by the characters '#', '$', or '%'. The '#' and '$' characters are used to prefix a full hexadecimal address. The '%' character adds **INT_MIN** (**MOSTNEG INT**) to the hexadecimal value using modulo arithmetic. This is useful when specifying transputer addresses which are signed and start at **INT_MIN**. For example, on a 32 bit transputer **%70** is interpreted as **#80000070** and on a 16 bit transputer as **#8070**.

15.6.1 Scrolling the display

Several commands mapped by the ITERM (see below) may be used to scroll certain of the Monitor page displays. Cursor keys may also be used.

15.6.2 Commands mapped by ITERM

Certain Monitor page commands are mapped to specific keys on the terminal by the ITERM file. Commands mapped in this way include keys which are used to scroll the display (see below), commands which produce the same effect in both debugging modes, and the commands RELOCATE and RETRACE which invoke the corresponding symbolic mode functions.

The keys to use for all Monitor page commands mapped by ITERM can be found by consulting the keyboard layouts supplied in the Delivery Manual.

15.6.3 Summary of main commands

Key	Meaning	Description
A	ASCII	View a region of memory in ASCII.
B ‡	Breakpoint	Display the Breakpoint menu enabling breakpoints to be set, cleared or listed.
C	Compare	Compare the code on the network with the code that should be there to ensure that the code has not been corrupted.
D	Disassemble	Display the transputer instructions at a specified area of memory.
E	Next Error	Switch the current display information to that of the next processor in the network which has halted with its error flag set.
F	Select file	Select a source file for symbolic display using the filename of the object file produced for it.
G	Goto process	Goto symbolic debugging for a particular process.
H	Hex	View a region of memory in hexadecimal.
I	Inspect	View a region of memory in any type. Types are expressed as occam types.
J ‡	Jump	Start or resume application program.
K	Processor names	Display the names of all processors in the network.
‡ = Breakpoint mode only		

Key	Meaning	Description
L	Links	Display instruction pointers and workspace descriptors for the processes currently waiting for input or output on a transputer link, or for a signal on the Event pin.
M	Memory map	Display the memory map of the current transputer.
N	Network dump	Copy the entire state of the transputer network into a 'network dump' file in order to allow continued (off-line) debugging at a later date.
O	Specify process	Resume the source level symbolic features of the debugger for a particular process.
P	Processor	Switch the current display information to that of another processor.
Q	Quit	Leave the debugger and return to the host operating system.
R	Run queues	Display instruction pointers and workspace descriptors of the processes on either the high or low priority active process queue.
S ‡	Show messages	Display the Messages menu enabling the default actions of the debugger to debug support functions to be changed.
T	Timer queues	Display instruction pointers, the workspace descriptors and the wake-up times of the processes on either the high or low priority timer queue.
U ‡	Update	Update the monitor page registers to reflect the current state of the processor.
V	Process names	Display the names of all processes on the current transputer.
W ‡	Write	Write to any portion of memory in any occam type (e.g. `REAL32`).
X	Exit	Return to symbolic mode.
Y ‡	Postmortem	Change a breakpoint debug session into a postmortem debug session.
?	Help	Display help information.
‡ = Breakpoint mode only		

15.6.4 Symbolic-type commands and scroll keys

Key	Description
TOP ♯ RETRACE ♯ RELOCATE ♯ HELP ♯ REFRESH ♯	Locate to the last instruction executed on the current processor. Switch to symbolic mode and perform symbolic operation. Switch to symbolic mode and perform symbolic operation. Display help information. Re-draw the screen.
LINE UP ♯ LINE DOWN ♯ PAGE UP ♯ PAGE DOWN ♯ ↑ ↓	Scroll the currently displayed memory, disassembly, or queue.
← →	Scroll the currently displayed processor left or right.
♯ For key bindings see the Delivery Manual.	

A ASCII

This command displays a segment of transputer memory in ASCII format, starting at a specific address. If no address is given the last specified address is used. Specify a start address after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press RETURN to accept the default (last specified) address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

The memory is displayed in blocks of 16 rows of 32 ASCII bytes, each row preceded by an absolute address in hexadecimal. Bytes are ordered from left to right in each row. Unprintable characters are substituted by a full stop.

↑, ↓, PAGE UP, PAGE DOWN keys can be used to scroll the display.

B Breakpoint menu (Breakpoint mode only)

This command invokes the Breakpoint Menu:

```
S - Set a breakpoint on this processor
T - Toggle a breakpoint on this processor
C - Clear a breakpoint
A - Clear all breakpoints on all processors
B - Clear all breakpoints on this processor
E - Set a breakpoint at all entries this processor
G - Set a breakpoint at all entries all processors
M - Set a breakpoint at all main () this processor
L - List all breakpoints
P - List all breakpoints on this processor
Q - Quit

Breakpoint option (A,B,C,E,G,L,M,P,Q,S,T) ?
```

Options are selected by entering one of the single letter commands. Pressing RETURN with no typed input when prompted for a breakpoint number or address cancels the option.

Breakpoints are assigned a unique number which must be specified with the 'C' option. Numbers are given on the List Breakpoints displays.

The 'E' and 'G' options which set breakpoints at the entrypoint of a process (at configuration level) are primarily intended for use with other INMOS language toolsets where there is no equivalent of a fixed name entrypoint (such as `main()` in C).

Note: Only breakpoints which are set in symbolic mode (at the beginning of a statement) are properly supported. Setting breakpoints at arbitrary addresses using the 'S' option may cause incorrect execution of the program.

C Compare memory

Compare memory compares the code on the network with the code that was loaded, to check that memory has not become corrupted.

Note: This option treats breakpoints as corrupted code.

The following menu is displayed:

```
      Compare memory
Number of processors in network is : 2

A - Check whole network for discrepancies
B - Check this processor for discrepancies
C - Compare memory on screen
D - Find first error on this processor
Q - Quit
```

Type one of the options A, B, C, D, or Q. Option 'Q' returns you to the Monitor page.

Checking the whole network – option A

Option 'A' checks the whole network processor by processor and displays a summary of the discrepancies found.

☐ If there no errors the following message is displayed:

```
Checked whole network OK
```

If any errors are detected the number of errors is given along with the address of the first error found and the name of the processor on which it occurred.

Checking a single processor – option B

Option 'B' checks just the current processor. In all other respects it is similar to option 'A'.

Compare memory on screen – option C

Option 'C' displays the actual and expected code for for each address in a block of memory. Discrepancies are marked with an asterisk ('*').

Memory is checked in blocks of 128 bytes. At the end of each block, type either 'Q' to quit, or SPACE to read and display the next block.

The format of the display is similar to the following example:

```
                Network Code        Correct Code
#800001234 : 0011223344556677  7766554433221100 *
#80000123C : 0011223344556677  0011223344556677
#800001244 : 0011223344556677  7766554433221100 *
   ...              ...              ...
#8000012AC : AABBCCDDEEFF0011  AABBCCDDEEFF0011

Press [DOWN] to scroll memory, [SPACE] for next
error, or Q to quit :
```

Pressing SPACE automatically invokes option 'D' – **Find first error....**

Find first error – option D Option 'D' searches the current processor's memory for the first occurrence of a discrepancy. If a discrepancy is found the display is switched to mode 'C' and the memory can be checked and displayed as in 'Compare memory on screen'.

D Disassemble memory

The Disassemble command disassembles memory into transputer instructions. Specify an address at which to start disassembly after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press RETURN to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '%h...h'.

The memory is displayed in batches of sixteen transputer instructions, starting with the instruction at the specified address. If the specified address is within an instruction, the disassembly begins at the start of that instruction. Where the preceding code is data ending with a transputer '**pfix**' or '**nfix**' instruction, disassembly begins at the start of the **pfix** or **nfix** code.

Each instruction is displayed on a single line preceded by the address corresponding to the first byte of the instruction. The disassembly is a direct translation of memory contents into instructions; it neither inserts labels, nor provides symbolic operands.

E Next Error

Next Error searches forward through the network for the next processor which has both its error and halt-on-error flags set. Processors are searched in the same order as they are listed by the '**K**' command, starting from the current processor and wrapping round. If a processor is found with both flags set the display is changed to the new processor as if the '**P**' option had been used. Press TOP to display the source line which caused the error.

If there is only one processor in the network you are informed of the fact.

F Select source file

This command selects a specified source file and invokes symbolic debugging. The full name of the object file (including extension) must be supplied.

This option allows breakpoints to be set in modules which have not yet been reached in the program's execution. (Source which has not yet been executed cannot be displayed using the 'O' or 'G' options because the `Iptr` and `Wdesc` addresses are not yet known.)

This option may also be used to browse source files rather like the CHANGE FILE symbolic function. However, unlike CHANGE FILE it allows you to use some of the symbolic debugging operations.

If a processor has been configured to contain different processes, this option first prompts for the process number of the source file:

```
Select process number (0 - N) ?
```

The range of numbers displayed in brackets are process numbers assigned by the debugger to different processes on the processor. Process names can be determined by using the Monitor page Process Name ('V') option before invoking the 'F' command.

Once a valid process number has been supplied (if applicable), the debugger prompts for the filename of the *compiled* object module. The full object filename (including extension) must be supplied.

```
Object module filename ?
```

The object filename must be specified because the debugger extracts the source code filename from the debug information in the compiled object file.

Note: At each prompt the command may be aborted by pressing RETURN with no typed input.

G Goto process

This command locates to the source code for any process which is currently shown on the screen. The cursor is positioned next to the **Iptr**, and permitted responses are listed on the screen as follows:

```
[CURSOR] then [RETURN], or 0 to F, (I)ptr,
(L)o, or (Q)uit
```

To select the desired process use the cursor keys to skip between processes on the screen, or specify a value 0 to F. Press RETURN to select the process indicated by the cursor. The saved **Iptr** is chosen by typing '**I**', and if currently in high priority, the interrupted low priority process is chosen by typing '**L**'. The sixteen processes shown on the right hand side of the display are chosen by typing '**0**' to '**F**'. Type '**Q**', FINISH, or REFRESH to abort this choice.

H Hex

The Hex command displays memory in hexadecimal. Specify the start address after the prompt:

```
Start address (#hhhhhhhh) ?
```

Press RETURN to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '**#**', or the short form '**%h...h**'. If the specified start address is within a word, the start address is aligned to the start of that word.

The memory is displayed as rows of words in hexadecimal format. Each row contains four or eight words, depending on transputer word length. Words are displayed in hexadecimal (four or eight hexadecimal digits depending on word length), most significant byte first.

For a four byte per word processor the sequence of bytes in a single row would be:

```
3 2 1 0   7 6 5 4   11 10 9 8   15 14 13 12
```

For a two byte per word processor, the ordering would be:

```
1 0   3 2   5 4   7 6   9 8   11 10   13 12   15 14
```

□ Words are ordered left to right in the row starting from the lowest address. The word specified by the start address is the top leftmost word of the display.

The address at the start of each line is an absolute address displayed in hexadecimal format.

I **Inspect memory**

The Inspect command can be used to inspect the contents of an entire array. Specify a start address after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press RETURN to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '`%h...h`'.

When a start address has been given, the following prompt is displayed:

```
    Typed memory dump
0 - ASCII
1 - INT
2 - BYTE
3 - BOOL
4 - INT16
5 - INT32
6 - INT64
7 - REAL32
8 - REAL64
9 - CHAN

Which type (1 - INT) ?
```

Give the number corresponding to the type you wish to display, or press RETURN to accept the default type. The types correspond to formal occam types as defined in the '*occam 2 Reference Manual*'. occam equivalences of C types are listed in the following table.

C type	occam type
int	INT
char, unsigned char	BYTE
short, signed short	INT16
long, signed long	INT32
float	REAL32
double, long double	REAL64

ASCII arrays are displayed in the format used by the Monitor page command 'ASCII'. Other types are displayed both in their normal representation and hexadecimal format.

The memory is displayed as sixteen rows of data. The address at the start of each line is an absolute address displayed as a hexadecimal number. The element specified by the start address is on the top row of the display.

Start addresses are aligned to the nearest valid boundary for the type, that is: **BYTE** and **BOOL** to the nearest byte; **INT16** to the nearest even byte; **INT**, **INT32**, **INT64**, **REAL32**, **REAL64**, and **CHAN** to the nearest word.

[J] Jump into and run program

This command starts up a program from the Monitor page, or restarts a process which has encountered a breakpoint or stop point inserted by the debug support functions **debug_assert()** and **debug_stop()**.

When starting a program the debugger converts (*patches*) the configuration external channels (those assigned to links) for each processor into *virtual* channels for use with the debugging kernel. This action is indicated by an activity indicator.

When the patching is complete the debugger prompts for a command line for the program:

```
Command line:
```

This is the command line used by the C runtime library to provide the **argc** and **argv** parameters to **main ()**.

When jumping into and resuming a program from a breakpoint, the following menu is displayed:

```
                Jump into Application

R - Resume breakpointed process
O - Resume all others
    (abandon breakpointed process)
J - Jump to different location
Q - Quit

Which option (J,O,Q,R) ?
```

When resuming from an error, the following submenu is displayed:

```
                Jump into Application

O - Resume all others
J - Jump to different location
Q - Quit

Which option (J,O,Q) ?
```

□ The four Resume options are listed in the following table.

Option	Description
R	Restarts the process that encountered the breakpoint.
O	Ignores the stopped process and resumes monitoring the network for other process activity. (When a process has stopped other processes continue to run until they either encounter a breakpoint or error, or become dependent on the stopped process.) **Note:** Using this option for a process stopped on a breakpoint removes the process forever.
J	Restarts the process from a different location. Only use this option if you are confident that the program can be resumed from the new location; resumption from most locations will corrupt the program.
Q	Quits the Resume submenu.

[K] **Processor names**

This command gives the processor numbers corresponding to processor names used in the configuration description. Processor numbers must be given when selecting specific processors for display by the debugger.

Note: The debugger displays only the first 19 characters of the processor name. If this is a problem you should make names unique within the first 19 characters.

[L] Links

The Links command displays the instruction pointer, workspace descriptor, and priority, of the processes waiting for communication on the links, or for a signal on the **Event** pin. If no process is waiting, the link is described as 'Empty'. Link connections on the processor, and the link from which the processor was booted are also displayed.

The format of the display is similar to the following example:

```
Link 0 out Empty
Link 1 out Empty
Link 2 out Iptr: #80000256 Wdesc: #80000091 (Lo)
Link 3 out Empty
Link 0 in  Empty
Link 1 in  Empty
Link 2 in  Iptr: #80000321 Wdesc: #80000125 (Lo)
Link 3 in  Iptr: #80000554 Wdesc: #80000170 (Hi)
Event  in  Empty

Link 0 connected to Host
Link 1 not connected
Link 2 connected to Processor 88, Link 1
Link 3 connected to Processor 1, Link 3

Booted from link 0
```

[M] Memory map

The Memory map command displays a memory map of the current processor. The display includes the address ranges of on-chip RAM, program code, configuration code, workspace and vectorspace, the sizes of each component in bytes rounded up to the nearest 1K bytes, total memory usage, and the address of 'MemStart', the first free location after the RAM reserved for the processor's own use.

Also displayed is the maximum number of processors that can be accommodated by the debugger's buffer space. This will depend on the amount of memory on the root processor, indicated to the debugger by the host environment variable `IDEBUGSIZE`.

The address of 'MemStart' is the value actually found on the transputer in the network. If this does not correspond to that expected by the configuration description, for example if a T414 was found when a T800 was expected, the following message is displayed:

```
MemStart should be : #80000070 (T800) !!!!!
```

If an incorrect MemStart is detected the symbolic functions may not work correctly. In these circumstances you should rebuild your program for the correct processor types on the network before reinvoking the debugger.

[N] Network dump

The Network dump command saves the state of the transputer network for later analysis. If you quit the debugger without creating a network dump file, debugging cannot continue from the same point without re-running the program. This is because the debugger itself overwrites parts of the memory on each transputer in the network.

Note: This command cannot be used in breakpoint mode (`idebug` command line option '`B`') or when post-mortem debugging a breakpoint session (`idebug` command line option '`M`').

Once a network dump file has been created, debugging can continue from the file, and the debugger does not need to be connected to the target network.

Before the dump file is created, the debugger calculates the disk space required, and requests confirmation. The size of the file depends on how much of each processor's memory is actually used in running the program, and is displayed as follows:

```
            Create network dump file
Number of processors to dump  : 2
File size excluding Freespace : 112604 bytes
File size including Freespace : 2097308 bytes

Continue with network dump (Y,N) ?
```

To continue with the network dump, type 'Y'.

You will them be prompted whether to include Freespace in the dump file (this is not normally required for configured programs).

```
Do you wish to include Freespace (Y,N) ?
```

Type 'Y' or 'N' as appropriate and specify a filename after the prompt:

```
Filename ("network.dmp", or "QUIT") ?
```

Press [RETURN] to accept the default filename, enter a filename (any extension will be replaced by '`.dmp`'), or type '`QUIT`' (uppercase) to exit.

□ If the file already exists, you are warned:

```
File "network.dmp" already exists
Overwrite it (Y,N) ?
```

If you type 'N', you are reprompted for the filename.

While the dump file is being written, a message is displayed at the terminal. For example:

```
Dumping network to file "network.dmp" ...
Processor 1 (T800)
Memory to dump : 10456 bytes ...
```

O Specify process

This command restores symbolic debugging, either at the same source line, or at another location. It can be used to locate to any source line, whether or not a process is waiting or executing there. To ensure the debugger locates to a valid process, it is better to use the 'G' command.

To return to symbolic debugging, the debugger requires values for `Iptr` and `Wdesc`. Specify `Iptr` after the prompt:

```
Iptr (#hhhhhhhh) ?
```

The default displayed in parentheses is the last line located to on this processor, or the address of the last instruction executed.

Either press RETURN to accept the default address, or enter the desired address. The address can be entered as a decimal number, a hexadecimal number preceded by '#', or the short form '`%h...h`'.

Useful addresses can be determined using the 'R', 'T', and 'L' commands to display specific addresses. The same addresses can be listed by using the 'G' command. The value of the saved low priority `Iptr` can also be used.

If the `Iptr` is not within the program body, the debugger indicates the type of code to which it corresponds.

After pressing any key you are returned to the Monitor page.

- If the **Iptr** is valid, you are prompted for the **Wdesc**:

```
Wdesc (#hhhhhhhh) ?
```

If a displayed **Iptr** was specified, its corresponding **Wdesc** is offered as a default. Press [RETURN] to accept the default, or specify a value in the same format as **Iptr**.

If no symbolic features other than a single 'locate' are required, then **Wdesc** is not needed and the default can be accepted.

If an invalid **Wdesc** is given, most of the symbolic features will not work, or will display incorrect values. However, you can still determine the values of scalar constants and some other symbols.

Any attempt to inspect or modify variables or channels, or to backtrace, will give one of the following messages:

```
Wdesc is invalid - Cannot backtrace

Wdesc is invalid - Cannot Inspect variables

Wdesc is invalid - Cannot Modify variables
```

If the location to be displayed is in a library for which the source is not available and the debugger cannot locate the call to that library, the following message is displayed:

```
Wdesc is invalid - Cannot auto backtrace out
of library
```

Once the **Iptr** and **Wdesc** have been supplied, the debugger displays the source code at the required location, and the full range of symbolic features are available.

P Change processor

This command changes to a different processor in the network. Specify the processor number after the prompt:

```
New processor number ?
```

To determine the mapping between the processor number and the processor name used in the configuration file, use the 'K' command. If the processor exists the display is changed to provide information about the specified processor. If the new processor's word length is different from that of the previous processor, the start address is reset to the bottom of memory.

If the processor is not in the configuration, the following message is displayed:

```
Error : That processor number does not
exist
```

To abort the command press RETURN with no input.

If there is only one processor in the network you are informed of the fact.

The cursor keys (← and →) can be used to scroll the list of processors. ← changes to the preceding processor and → to the next processor in the sequence. The processor sequence is the same as that displayed by the 'K' command.

Q Quit

This command quits the debugger and returns to the operating system. Once quit, the debugger cannot be used to debug the same program without reloading the program unless a 'network dump' file has been created. This is because using the debugger overwrites some of the contents of the network.

R Run queues

This command displays **Iptrs** and **Wdescs** for processes waiting on the processor's active process queues. If both high and low priority front process queues are empty, the following message is displayed:

```
Both process queues are empty
```

If neither queue is empty, you are required to specify the queue:

```
High or low priority process queue ? (H,L)
```

Type '**H**' or '**L**' as required. If only one queue is empty, the debugger displays the non-empty queue.

The screen display is paged. To view other processes scroll the display using the CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN, PAGE UP, and PAGE DOWN keys.

Note: In breakpoint mode this command may provide incorrect results because the queues may include processes which form the debugging kernel. An asterisk next to the queue heading indicates where this is so.

S Show debugging messages

This command is used to enable and disable debugging messages and prompts. It invokes the following submenu:

```
                Show Messages Menu

B -- Show message for breakpoints : ON
D -- Show debug messages          : ON
E -- Show message for errors      : ON
Q -- Quit

Which option (B,D,E,Q) ?
```

Options **B** and **E** control the display of prompts when a breakpoint or error (via the library functions **debug_assert** and **debug_stop**) is encountered. Disabling these options ensures that the debugger is entered on a breakpoint or error without requesting confirmation.

Option **D** controls the display of debugging messages inserted with the **debug_message** library function.

[T] Timer queues

This command displays `Iptrs`, `Wdescs`, and wake-up times for processes waiting on the processor's timer queues. Prompts and displays are similar to those for the Run queue command.

[TOP] Last Instruction

This command is used to display the source corresponding to the last instruction to be executed on the current processor. It is the same as typing '`G`', then '`I`'.

[U] Update registers

This command updates the clock and status display (e.g. runtime queues) for the current processor. It enables you to monitor the activity of other processes while one process is stopped at a breakpoint or error.

[V] Process names

This command gives the process numbers corresponding process names used in the configuration description. Process numbers must be given when selecting specific processes for display by the debugger.

Note: The debugger displays only the first 19 characters of the process name. If this is a problem you should make names unique within the first 19 characters.

[W] Write to memory

This command writes a value to a specified address. Values must be specified in the current type (the type used in the previous Monitor page Inspect command), or `INT` if the type was a `CHAN` or the Disassemble or Hex options have been used after an Inspect.

[X] Exit

This command returns to symbolic mode and locates to the current address.

[Y] Enter post-mortem debugging

This command allows the debugger to be switched into post-mortem mode when the program crashes (a process sets the error flag on any processor). Halted processors prevent the breakpoint debugger from accessing the network correctly and debugging must continue in post-mortem mode.

If the program has not crashed, the debugger prompts for confirmation:

```
The program has not crashed - are you sure (Y,N) ?
```

If you have disabled checking of the subsystem error status (the command line 'S' option), you are prompted with:

```
Unable to detect if the program has crashed -
are you sure (Y,N) ?
```

Typing 'Y' continues the operation, typing 'N' aborts it.

This command performs the same action as quitting the debugger when in breakpoint mode and restarting it using the 'M' command line option instead of 'B'.

Note: State information for a process that has stopped (on breakpoint or error) will be lost when switching from breakpoint to post-mortem mode. If the information is important you should make a note of it before switching modes.

15.6.5 Symbolic-type commands

TOP	This command locates to the last instruction executed on the current processor.
RELOCATE	This command returns to symbolic mode and performs a symbolic RELOCATE. It cannot be used if the processor has been changed at the Monitor page.
RETRACE	This command returns to symbolic mode and performs a symbolic RETRACE. It cannot be used if the processor has been changed at the Monitor page.
? HELP	These commands display a summary of the commands available at the Monitor page.
REFRESH	This command refreshes the screen.

15.7 Symbolic functions

Symbolic debugging allows high level language programs to be debugged from the identifiers used in the source code. Symbolic identifiers are the names given in the program to variables, constants, channels, and functions.

Symbolic functions are invoked using keyboard function keys. Keyboard layouts for common terminal types can be found in the rear of the Delivery Manual that accompanies the release.

Symbolic debugging functions are listed in Table 15.3. Functions only available in breakpoint mode are marked with a double dagger (‡).

Function	Description
INSPECT	Display the value and type of a source code symbol.
CHANNEL	Locate to the process waiting on a channel.
TOP	Locate back to the error, or last source code location.
RETRACE	Retrace the last BACKTRACE etc.
RELOCATE	Locate back to the last location line.
INFO	Display extra process information.
MODIFY ‡	Change the value of a variable in memory.
RESUME ‡	Resume the application program from the breakpoint.
MONITOR	Change to the Monitor page.
BACKTRACE	Locate to the procedure or function call.
HELP	Display a summary of utility key uses.
GET ADDRESS	Display the location of a source line in memory.
GOTO LINE	Go to a specific line in the file.
SEARCH	Search for a specified string.
ENTER FILE	Change to an included file.
EXIT FILE	Change to an enclosing file.
CHANGE FILE	Display a different source file.
TOP OF FILE	Go to the first line in the file.
BOTTOM OF FILE	Go to the last line in the file.
TOGGLE BREAK ‡	Set or clear a break on the current line.
INTERRUPT ‡	Force the debugger into the Monitor page without stopping the program.
CONTINUE FROM ‡	Resume the application program from the current line.
TOGGLE HEX	Enables/disables Hex-oriented display of constants and variables.
FINISH	Quit the debugger.
‡ = Breakpoint mode only	

Table 15.3 Debugger symbolic functions

INSPECT This function allows you to determine the value and type of variables and constants, and provides useful information about other source code symbols such as functions and channels. To inspect a symbol, place the cursor on the name and press INSPECT, or press INSPECT and give a symbol name. Spaces and the case of the letters in the name are significant. Specifying an empty expression aborts the INSPECT operation.

The symbol must be in scope with the line to which the debugger last located, which may not be the same as the current cursor position.

Expressions for variables can incorporate numbers, pointers, and any in-scope identifiers. For pointers to functions an address is displayed which can be used to locate to the source code using the Monitor page 'O' command. For semaphores, pointers to processes waiting on the semaphore can be determined from the display of its data structure. For enumerated types their symbolic names are also displayed.

Details of the display formats for all variable types are given in section 15.9.

Expression language

INSPECT supports an expression language for variables which follows C syntax but with some limitations and extensions (for example, subranging of arrays is supported). The MODIFY command also supports the same language.

Details of the expression language can be found in section 15.8.

CHANNEL This function jumps down a channel if a process is waiting at the other end. Use this key as you would INSPECT, but when positioned on a channel. The debugger locates to the source line corresponding to the waiting process from where the process can be debugged. This function is invalid if the cursor is not on a channel or the name specified is not a channel.

The CHANNEL function allows you to 'jump' to other processors along transputer links. If a process running on another processor is waiting for communication on a channel the debugger 'jumps down' the link and automatically changes to that processor.

TOP This function locates back to the line containing the original error, or to the line located to by the previous invocation of the Monitor page 'G' or 'O' command.

RETRACE This function locates back to the previous location. Repeated use of RETRACE reverses the effect of successive BACKTRACE, CHANNEL, and TOP operations.

RELOCATE This function locates back to the last point located to by the debugger. For example, it can be used to return to the original source line of an error after browsing the code with the cursor and scroll keys.

INFO This function displays the `Iptr` and `Wdesc` of the last location, the process name and priority, and the processor number.

If the `Wdesc` is not in the defined region for a process the message: `Undefined process` is displayed in place of the process name. For single processor programs that have not been configured there is no defined region and the message: `Stack area unknown` is displayed to reflect this.

If a `Wdesc` has not been supplied, it is given as '`invalid`'.

SEARCH This function searches forwards in the source file for a specific string. Either specify a search string or press RETURN to accept the default, which is the last string specified.

HELP This function displays a brief summary of the debugger symbolic function keys.

MONITOR This function recalls the Monitor page environment.

FINISH This function quits the debugger. The Monitor page 'Q' option has the same effect.

BACKTRACE This function locates to the line where a procedure or function was called. If the debugger is already located in the program's main procedure, no backtrace is possible and the following message is displayed:

```
Error : Cannot backtrace from here
```

GET ADDRESS	This function displays the address of the transputer code which was compiled for the source line where the cursor is currently placed.
CHANGE FILE	This function opens a different source file for reading only. No symbolic functions are available, unlike the Monitor page 'F' option.
TOGGLE HEX	This function displays Hex values of C variables as well as their decimal values. The default is to display integral types in decimal format only.
INTERRUPT	This function forces the debugger to enter the Monitor page without stopping the program. **Note:** This command does not operate if there are keystrokes waiting before it in the keyboard buffers. It may also fail if the application program is waiting for input from the keyboard.
ENTER FILE	Enters an included file. Position the cursor on the relevant `#include` directive and press ENTER FILE.
EXIT FILE	Exits from an open included file.
GOTO LINE	This function allows you to change to a particular line in the source. Specify a line number, or type 0 (zero) to abort the operation.
TOP OF FILE	Moves to the start of the file.
BOTTOM OF FILE	Moves to the end of the file.

15.7.1 Breakpoint functions

TOGGLE BREAK This function toggles a breakpoint on the source line indicated by the cursor and provides information on the breakpoint number (as used by the Monitor page 'B' command), whether it was set or cleared, and the line number it is on.

When the source line the cursor is on produces no associated object code the debugger displays an exclamation mark (<!>) after the line number to indicate that the breakpoint has been toggled on a different line to the one the cursor is on (as shown at the bottom of the display).

RESUME This function restarts the program from the breakpoint. (To restart from an error use CONTINUE FROM).

CONTINUE FROM This function restarts the program from the line indicated by the cursor. CONTINUE FROM should only be used to bypass an erroneous source line. The result of continuing from other points in the code may be unpredictable if there are intervening stack adjustments.

MODIFY This function changes the value of a variable in transputer memory. Like INSPECT this function accepts expressions involving any symbol in scope. To modify a variable place the cursor on the name and press MODIFY. The debugger then prompts for the *destination* followed by the *source*, which can both be given as *expressions*. The *destination* expression is the variable or constant you wish to change; the *source* expression is the new value that will be assigned. Specifying an empty expression aborts the MODIFY operation.

Variables are specified using the same expression language that is used by INSPECT. The language uses the syntax of C but with some limitations. A description of the language can be found in section 15.8.

15.8 Expression language for INSPECT and MODIFY

The expression language for source code symbols (variables, constants, and channels) follows the syntax of C with some minor modifications.

Limitations and extensions to C syntax are described in the following sections.

15.8.1 C syntax not supported

The following table summarises aspects of C syntax not supported in the expression language.

Area of limitation	Example of limitation
Casting to pointer types	`(char *) ptr`
Address operator `&` returns `int` rather than a pointer to the type	`&baz`
Calling of functions	`sqrt (x)`
Input of strings	`"a string"`
Input of initialiser lists	`{ 1, 2, 3 }`
Trigraph sequences	`'??('`
Bit field modification	
Modification using assignment	`x = y`
Conditional operator	a ? b : c

15.8.2 Extensions to C syntax

The language supports the specification of array subranges for *arithmetic* data types. Subranges are specified as two array bounds separated by a semicolon. For example: `foo[2;4]` displays the values of elements `foo[2]`, `foo[3]` and `foo[4]`.

Note: For arrays and structures the information displayed will normally overwrite part of the screen display. Press any key when prompted to restore the display.

15.8.3 Editing keys

The following editing functions are available for on-screen editing of expressions:

Key	Effect
START OF LINE	Move the cursor to the beginning of the expression.
END OF LINE	Move the cursor to the end of the expression.
DELETE LINE	Delete the expression.
←	Move the cursor left one character.
→	Move the cursor right one character.
↑	Replace the current expression with the expression used in the previous INSPECT or MODIFY.
DELETE	Delete the character to the left of the cursor.
RETURN	Enter the expression for evaluation.

Note: START OF LINE, END OF LINE, DELETE LINE, and DELETE are mapped by the ITERM file to specific keys on the keyboard. Details of the key mappings on your terminal can be found in the Delivery Manual that accompanies the release.

15.8.4 Types

C types are interpreted and displayed by the debugger as follows:

Name	Member types
Character	`char`, `signed char`, `unsigned char`
Floating	`float`, `double`, `long double`
Basic	Character, `signed integer`, `unsigned integer`, Floating
Integral	Character, `signed integer`, `unsigned integer`, `enum`
Arithmetic	Integral, Floating
Scalar	Arithmetic, Pointer
Derived	Array, Function, Pointer, `struct`, `union`

Type compatibility when using MODIFY

Source and destination expressions must be type compatible according to the rules of C. Scalar types are cast automatically into other scalar types but non-scalar expressions must be strictly compatible.

Type conversions, where required, are performed according to normal C promo-

tion rules.

The following examples illustrate the rules governing type compatibility.

Given the following declarations:

```
int two_d_array[2][10];
int one_d_array[10];
int foo;
char bar;
```

the following modifications are permitted:

Source:	`one_d_array`	(array of 10 integers)
Destination:	`two_d_array[1]`	(row of 10 integers)

Source:	`foo`	
Destination:	`bar`	

Source:	`two_d_array[1][2]`	(single element)
Destination:	`bar`	(single integer)

The following modification is not permitted:

Source:	`two_d_array[1]`	(row of 10 integers)
Destination:	`foo`	(single integer)

15.9 Display formats for source code symbols

When displaying an object, **idebug** (where possible) will also display type information for an object (e.g. **unsigned char**).

15.9.1 Warnings

When evaluating an expression, checking is performed which may lead to warning messages being produced (eg. overflow in arithmetic operation). Such warnings are intended to highlight potential problems and to ensure that a user understands any action **idebug** is taking.

15.9.2 TOGGLE HEX key

This key enables Hex Integer Print to be toggled.

tt idebug attempts to display *integral* types in the format it believes is most appropriate.

This means that by default, integer values (including enumerated types) are displayed in decimal, addresses are displayed in Hex and decimal, and characters are printed in decimal along with the corresponding character constant. By use of TOGGLE HEX, the default behaviour may be overridden to cause **idebug** to print in Hex and decimal for integral types, and in decimal with the corresponding Hex character constant for characters.

15.9.3 Notation

In the following descriptions, the following notation is used:

ddd	indicates a (possibly signed) decimal value
'c'	indicates a character
0xHHH	indicates a hexadecimal value
'\HH'	indicates a hexadecimal character
fff	indicates a floating point number of the form: **ddd.ddd or ddd.dddEddd**
type	indicates the type of the object
" "	indicates a character string in an array
" "...	indicates a character string of unknown length which is terminated by a null character (which is not shown)
{ }	indicates a list
{ }...	indicates a character list of unknown length which is terminated by a null character (which is shown)
< >	indicates the contents of a basic or channel object when a pointer points to it (except when the object is volatile)
( )	provides extra information about an object

15.9.4 Basic Types

Display formats for basic C types are given in the Table 15.4. Displays are given in normal decimal format and in Hex format (invoked by TOGGLE HEX).

15.9.5 Enumerated types

Variables of an enumerated type are displayed as their integer value (in exactly the same manner as an **int**) followed by the name of the enumeration and

Type	Hex Integer Print Off	Hex Integer Print On
`char`	`ddd ('c')` *type*	`ddd ('\xHH')` *type*
`short`	`ddd` *type*	`0xHHH (ddd)` *type*
`int`	`ddd` *type*	`0xHHH (ddd)` *type*
`long`	`ddd` *type*	`0xHHH (ddd)` *type*
`float`	`fff` *float*	`fff` *float*
`double`	`fff` *double*	`fff` *double*
`long double`	`fff` *double*	`fff` *double*

Table 15.4 Display formats – basic C types

the enumeration constant name for the value. If there are multiple enumerated constants that share the same value, a list is formed containing all of the enumeration constant names. `invalid enum constant` is used to indicate when a value is not a member of an enumerated type.

integer `(`*enum-tag-name*`:` *enum-const-name*`)`
integer `(`*enum-tag-name*`:` `{`*enum-const-name, ...*`})`
integer `(`*enum-tag-name*`:` `invalid enum constant)`

15.9.6 Pointers

Pointers are displayed in one of the following ways:

```
0xHHH (null pointer)
0xHHH (pointer to const volatile)
0xHHH (pointer to volatile)
0xHHH (channel pointer to link)
0xHHH (channel pointer)
0xHHH * (mis-aligned pointer)
0xHHH < basic or channel object >
0xHHH (pointer)
```

15.9.7 Function Pointers

If the function pointed to is defined in the current module the name of the function is displayed.

```
0xHHH (function pointer to "functionname ()")
0xHHH (cannot find corresponding function)
```

15.9.8 Structs

In order to display structures in a readable manner, members which are derived types are not always displayed in as much detail as when the member occurs on its own. To obtain more detail select the member of the structure explicitly.

identifier = {

```
        member1 = object
        member2 = object
        member3 = object
            .
            .
}
```

15.9.9 Unions

Unions are displayed in the same manner as structs except that a question mark ? appears to the left of each member to indicate that only one member of the union should be selected.

identifier = {

```
   ?    member1 = object
   ?    member2 = object
   ?    member3 = object
   ?        .
   ?        .
}
```

15.9.10 Addressof (&) operator

The result of the addressof & operator is automatically printed as a Hex and integer value regardless of the setting of TOGGLE HEX.

Note: that the result type of addressof & is an `int` rather than a pointer to the type used.

&*identifier* = `0xHHH (ddd)`

15.9.11 Arrays

When displaying arrays `idebug` prints the valid range of each dimension (if known) in addition to any type information and the contents of the array.

For single dimension arrays containing *arithmetic* types each member of the array is displayed regardless of the size of the array. For large arrays **idebug** requests confirmation to continue during the display.

For large arrays where the full display may be unwieldy use array subranging to display the area of interest.

identifier = **array [0..M] of** *list of arithmetic type*
identifier = **array [0..M] of** *type*
identifier = **array [0..M][0..N] of** *type*

15.9.12 Channels

Channels are displayed to show information on the process that is waiting at the other end (or **Empty** if no is process waiting).

identifier = **Channel <Iptr:** *address*, **Wdesc:** *address* **(Lo)>**
identifier = **Channel <Iptr:** *address*, **Wdesc:** *address* **(Hi)>**
identifier = **Channel <Empty>**

An asterisk (*) is used to denote when an **Iptr** or **Wdesc** value is not within the defined memory map range of the program (i.e. the value is invalid).

identifier = **Channel <Iptr:** *address* ***, Wdesc:** *address* ***>**

15.10 Example displays

Consider the following source code segment compiled for a 32 bit transputer (for a 16 bit transputer integers in Hex format and addresses would contain 4 Hex digits instead of 8):

```
enum colour { red = 1 };
struct Many   {
        int     a;
        double  b;
};

enum colour     shoe = red;
struct Many     many = { -42, 2.0 };

int     answer   = 42;
char    key      = 'A';
char    string[] = "bye";
char*   ptr      = string;
short   iarray[] = { 1, 2, 3, 4 };
```

Expression	Display (Hex Integer Print Off)
answer	answer = 42 int
&answer	&answer = 0x801FFF2C (-2145386708) int
key	key = 65 ('A') unsigned char
string	string = array [0..3] of unsigned char "bye\0"
ptr	ptr = 0x801FFF18 < "bye"... unsigned char >
iarray[1;2]	iarray[1;2] = subarray of short {2, 3}
shoe	shoe = 1 int (colour: red)
red	red = 1 int (enum constant)

many	many = {
	a = -42 int
	b = 2.0 double
	}

Expression	Display (Hex Integer Print On)
answer	answer = 0x0000002A (42) int
&answer	&answer = 0x801FFF2C (-2145386708) int
key	key = 65 ('\x41') unsigned char
string	string = array [0..3] of unsigned char {\x62, \x79, \x65, \x00}
ptr	ptr = 0x801FFF18 < {\x62, \x79, \x65, \x00}... unsigned char >
iarray[1;2]	iarray[1;2] = subarray of short {0x0002, 0x0003}
shoe	shoe = 0x00000001 (1) int (colour: red)
red	red = 0x00000001 (1) int (enum constant)

many	many = {
	a = 0xFFFFFFD6 (-42) int
	b = 2.0 double
	}

15.11 Error messages

This section lists errors generated by **idebug**. Other messages not in this list may be generated by corrupt files and by files not created by the toolset.

15.11.1 Out of memory errors

If the debugger runs out of memory when trying to read in information and the offending item cannot be reduced in size, the amount of memory available to the debugger may be increased by increasing the size of the memory on the transputer the debugger is running on and updating **IDEBUGSIZE** accordingly.

15.11.2 If the debugger hangs

If the debugger starts up but then hangs with the message:

```
Loading network...
```

either of the following errors may have occurred:

1 The network connectivity is not correctly described in the configuration description, for example, a link is not connected to a processor, or the type of a processor has been specified incorrectly.

   Network connectivity on a board can be checked by running a check or worm program, such as the **ispy** program supplied with the board support software for INMOS *i*q systems products. These products are available separately from your local INMOS distributor.

2 You have set **IDEBUGSIZE** to be larger than the memory on the processor where the debugger is running.

   Change **IDEBUGSIZE** to reflect the correct memory size.

15.11.3 Error message list

"*filename*" not compiled with full symbolic debug information

The object code module does not contain sufficient debug information for the debugger to locate to its corresponding source code (i.e. it contains minimal debug information). Recompile the module and rebuild the program in order to debug it symbolically.

Already located - No process is waiting at the other end of this link

An attempt to jump down a hard channel (link) has failed because there is no process waiting at the other end.

Attempted read outside Parameter block
Attempted write outside Parameter block

The configuration system has become corrupted. Check hardware using a memory check program such as `ispy`. (The `ispy` program is supplied as part of the board support software for INMOS *i*q systems products. These products are available separately from your local INMOS distributor.)

Can only specify a transputer type if bootable is for a class

You have tried to specify a processor type when the bootable file is already for a specific processor type.

Cannot create network dump - *reason*

Creation of a network dump file is not permitted on a program that is, or has been, breakpointed.

reason can be either of the following:

1 **Not for breakpoint postmortem** – invalid when post-mortem debugging a breakpoint debug session.

2 **Not while breakpointing** – invalid in breakpoint mode.

Cannot find this line's location

Either of the following has occurred:

1 You have moved the cursor beyond the end of the current source file for which there is no executable code.

2 The compiler has optimised the executable code out.

Cannot locate beyond Freespace area

The address specified is not within the memory map range of the processor.

Cannot locate to *area* (Iptr: #*address*)

The address specified is not within the **code** area for the program on the processor. *area* can be any of the following:

Reserved transputer memory
Runtime kernel
Configuration code area
Vectorspace area
Static area
Heap area
Freespace area

Cannot open "*filename*"

Either the file does not exist or it is not on the **ISEARCH** path (note that by default this includes the current directory). The **ilist** tool can be used to confirm this.

Cannot read processor *number* (T*xxx*)

The debugger cannot communicate with that processor. Any of the following errors may have occurred:

1) The root processor's core dump has been incorrectly specified.

2) The debugger has failed to analyse the network correctly. Either you have failed to specify the '**A**' option or the system control signals are wired incorrectly.

3) The network does not match that specified in the configuration file. Check network connectivity using a check program such as **ispy**. (The **ispy** program is supplied as part of the board support software for INMOS *i*q systems products. These products are available separately from your local INMOS distributor.)

Cannot run application – the program has crashed !

Use the '**Y**' (Enter post-mortem debugging) command to post-mortem debug the (now defunct) breakpoint session.

Channel is invalid

The channel does not point to a known process executing on the processor.

Debug info too large (*reason*)

The debugging information for a particular compilation module is too large for the debugger. Either reduce the size of the offending module or increase the size of memory on the processor where the debugger is running (see section 15.11.1 on how to overcome this).

reason can be any of the following:

lx.tags is full
ws.array is full
name table is full

Debugger incompatible configuration file "*filename***"**

You have configured your program without specifying the debugger compatible option ('G' option) to the configurer (this option disables code segment re-ordering).

Debugger incompatible ROM configuration file "*filename***"**

You have configured your program to be ROM-loadable. The debugger can only debug bootable programs.

Duplicate debugger modes: *message*

Mutually incompatible options have been specified on the command line.

enum constant *name* **is not in scope**

The identifier *name* exists in the module, but is not in scope from where the debugger last located to. In order to inspect the identifier you must locate to a new position where the symbol is in scope.

File has changed since configuration "*filename***"**

You should rebuild the program again.

File was not included

The `#include` line on which you are trying to use ENTER FILE was not included by the compiler (probably as a result of a preprocessor conditional action).

FILE IS TOO BIG - truncated

The debugger buffer capacity has been exceeded. The buffer contains as much of the file as could be read before the capacity was exceeded (see section 15.11.1 on how to overcome this).

Illegal virtual channel address

The channel has been (possibly incorrectly) tagged as virtual but does not point to a valid virtual channel. This is caused by channel pointers that have either not been initialised or have become corrupted.

Interactive debugging has disabled

The module has been linked with the linker '**Y**' option to disable breakpoint (interactive) debugging. Rebuild your program without disabling interactive debugging and retry.

ITERM error on line *linenumber, message*

The debugger has detected a syntax error in the ITERM file. *message* describes the error.

Linker complains that any of the following debug support functions are not found:

debug_assert ()
debug_message ()
debug_stop ()

You have omitted the **#include <misc.h>** directive required for the debug support functions.

No need to assert Subsystem Analyse

The '**A**' option is not required when you specify options '**N**' or '**D**'.

Not a (compatible) bootable file *"filename"*

The file is either a non-bootable file or a pre-product release bootable file. Use **ilist** to determine the contents of the file if in doubt.

Not enough free memory for the debugger

You have either not set the environment variable **IDEBUGSIZE** or you

have set it to be too small (it should be > 400K).

Change the variable to reflect the memory size of the root processor.

Not on a valid #include line

You may only use ENTER FILE when the cursor is on a line with a `#include` directive.

Only debugging tools and cursor keys are available

You have pressed a key which is not defined.

Option must be followed by a link number (0 - 3)

Options '`B`', '`M`', and '`T`' require a link number in the range 0 - 3.

Option must be followed by a valid Processor type (eg. T425)

The processor type supplied is not recognised by the debugger.

(Probe Go) : Processor *number* **- Cannot contact**

The debugger is unable to communicate with processor *number*. The processor type specified in the configuration (or to the debugger via the '`C`' option) does not match that found. Check the network using a program such as `ispy` in order to determine the correct processor type. (The `ispy` program is supplied as part of the board support software for INMOS *i*q systems products. These products are available separately from your local INMOS distributor.)

(Probe Go) : Processor *number* **- Invalid processor type**

The processor type specified in the configuration (or to the debugger via the '`C`' option) does not match that found. Check the network using a program such as `ispy` in order to determine the correct processor type.

(Probe Resume) : Processor *number* **- Invalid Breakpoint**

The debugger has stopped at a breakpoint which it did not place in the code. If you wish to continue executing the program set a breakpoint at the same address and retry the command.

Processor *number*: **Insufficient memory, require at least** *number* **bytes**

The memory requirement of the processor as specified to the configurer,

collector, or in **IBOARDSIZE** (as appropriate) is too small. (Note that the value displayed may include memory for some configuration code that is reclaimed when program starts executing).

This may also be caused by the debugging Runtime kernel using an extra 10-14K of memory.

Processor type must be a 32 bit processor (eg. T425)

You must specify a 32 bit processor type because processor classes are for 32 bit processors only.

Processor type must be not abbreviated

You must specify specific processor types rather than abbreviated types (e.g. T425 rather than T5) because some abbreviated types cover more than one specific type.

READ ERROR - truncated

The debugger could not read all of the file. The buffer contains as much of the file as could be read (see section 15.11.1 on how to overcome this).

Runtime kernel is not present (or has been overwritten)

Either the runtime kernel has been corrupted or you are trying to post-mortem a breakpoint session that didn't occur.

There is no enclosing #include

You have attempted to use EXIT FILE when not located in a nested include file.

There are no processes waiting at either end of this link

An attempt to jump down a hard channel (link) has failed because there are no processes waiting at either end.

This transputer link is connected to the host

The link specified in the '**B**', '**M**', and '**T**' option is the communication link from the debugger to the host and is not connected to the network.

Too many processes used at configuration level (*number*)

The debugger requires more memory in order to operate on this many processes (see section 15.11.1 on how to overcome this).

Too many processors - There is only enough room for (*number*)

The debugger requires more memory in order to operate on this many processors (see section 15.11.1 on how to overcome this).

Unable to find file line entry in debug info

This error may occur when trying to ENTER an include file, if conditional preprocessor directives are present (e.g. `#ifdef`) which cause certain lines to be excluded from the compilation, and the debugger is not aware of this.

Unable to read environment variable ITERM

There is no translation for the ITERM environment variable which defines the screen and keyboard format.

Unable to toggle a breakpoint on this line

The breakpoint cannot be set or cleared on this source line. Either:

1 The current file contains no executable code or

2 Executable code is contained in an include file and the debugger cannot determine whether you mean to toggle the breakpoint in that file or in the current file.

Move to the line where you really want to toggle the breakpoint and retry the command.

Unknown core dump format filename

The network dump file is in the wrong format or the wrong file has specified.

variable *name* is not in scope

The identifier *name* exists in the module, but is not in scope from where the debugger last located to. In order to inspect the identifier you must locate to a new position where the symbol is in scope.

Wdesc is invalid - *message*

The `Wdesc` supplied is invalid: this may be deliberate because it is unknown. If you supplied it from the Monitor page environment, retry the command with a valid `Wdesc`.

message can be one of:

cannot inspect variables
cannot modify variables
cannot backtrace
cannot auto backtrace out of library

Wrong number of processors in network dump file filename

The number of processors does not correspond to the current program. The wrong network dump file may have been specified.

You cannot backtrace from here

The procedure you are trying to backtrace was called by the program's bootstrap routine which cannot be accessed by the debugger.

You have changed file, so you can't use this key

There are certain symbolic features that you may not do if you have changed file. Either press RELOCATE before retrying the command or relocate to the file from the Monitor page using the 'F' (Select file) command.

You must specify a filename

The command line syntax requires a filename.

You must specify the application boardsize in IBOARDSIZE to be <= #10000

On a T2 the maximum memory size is 64K (#10000).

16 idump – memory dumper

This chapter describes the memory dumper tool **idump** that dumps the contents of the root processor's memory to disk. It is used to enable the debugging of code running on the root transputer.

16.1 Introduction

The memory dumper allows programs that use the root transputer to be debugged in the normal way using the debugger tool **idebug**. It is required because **idebug** runs on the root transputer and overwrites all code and code in its memory.

idump saves the contents of the root transputer to a disk file in a format that can be read by the debugger. Information contained in the file allows the debugger to analyse data in the root transputer in the same manner as other transputers on the network.

When **idump** is invoked it calls the server with the '**SA**' option so that the space occupied by the dumper program is saved before it is loaded onto the transputer.

16.2 Running the memory dumper

To invoke the **idump** tool, use the following command line:

▶ **idump** *filename memorysize* {*startoffset length*}

where: *filename* is the name of the dump file to be created.

memorysize is the number of bytes, starting at the bottom of memory, to be written to the file.

startoffset is an offset in bytes from the start of memory.

length is the amount of memory in bytes, starting at *startoffset*, to be dumped in addition to *memorysize*.

All parameters can be expressed in either decimal or in hexadecimal format. Hexadecimal numbers must be preceded by the hash # character or the dollar sign $.

The memory dump file stores the contents of the transputer's registers and the first *memorysize* bytes of memory. The file is given the `.dmp` extension. After the dump has been performed `idump` remains resident on the transputer board ready to load the debugger.

memorysize must be large enough to contain the complete program with its workspace and vectorspace. If the program to be dumped uses the free memory buffer, the whole of the transputer board's memory should be dumped.

Further portions of memory can be dumped by specifying the start of the segment of memory to be dumped and the number of bytes, using pairs of *startoffset length* parameters. The start address is given by *startoffset* and the number of bytes by *length*.

The overall size of the memory dump file is given by the amount of memory saved plus around 500 bytes for the register contents.

16.2.1 Example of use

Assuming an `IBOARDSIZE` of 100K:

```
idump core 100000
```

16.3 Error messages

Badly formed command line

Command line error. The command syntax requires a file name followed by the number of bytes of memory to dump. Check the syntax of the command and retry.

Cannot open file

File system error. The memory dump file could not be opened on the host system.

Cannot write file

File system error. The memory dump file could not be written to the host system.

You must tell the server to peek the transputer

`idump` has been invoked by calling the host file server with the incorrect option. This error can only occur if the tool is not invoked with the supplied executable file `idump.exe`.

17 `iemit` – memory configurer

This chapter describes the Memory Configuration tool **`iemit`**. This tool can be used interactively to enable the user to explore the effects of changes in the memory interface parameters of certain 32 bit transputers. The tool can also be used in batch mode to create ASCII or PostScript files. The tool produces a memory configuration file which may be included as an input file to **`ieprom`** and blown into EPROM along with a ROM-bootable application file.

The chapter describes how to use **`iemit`** and outlines its capabilities. Example displays are provided followed by a list of error messages which the tool may generate. The format of the memory configuration file is described and an example is given. **Note**: memory configuration files are simple text files which may be created manually using a standard editor or generated by using **`iemit`**.

Finally the chapter describes a tool called **`icvemit`**. This tool is provided to convert memory configuration files produced by **`iemi`** (a previous version of **`iemit`**), to the file format recognised by the current release of **`iemit`** and **`ieprom`**. The command line syntax is described and a list of possible error messages is given.

17.1 Introduction

The IMS T400, T414, T425, T800 and the T805 transputers have a configurable external memory interface which allows a variety of types of memory device to be connected using few extra components.

For these transputers, the interface configuration may be selected by one of two mechanisms. The user may select one of the 17 standard memory configurations (13 for the T414) or a customised memory configuration may be loaded from a ROM or PAL on reset.

Both methods of memory configuration are available when booting from ROM or from link. If the transputer is being booted from ROM, a customised memory configuration may be added to the ROM or a standard configuration may be used. If the transputer is booted from link a standard configuration may be used at no extra cost, or a dedicated ROM or PAL may be added for a customised configuration.

In order to generate a customised configuration the user may create a memory configuration file, describing the memory configuration and have this blown into an EPROM. The configuration chosen is made known to the transputer by

simple board level connections which are detected by the transputer on reset. If a standard configuration is required the **MemConfig** pin is connected to the appropriate address pin. For example, standard configuration 7 is selected via address pin **MemAD7**. If a customised configuration is required the **MemConfig** pin is connected though an invertor to the appropriate data line, usually this is **MemnotWrD0**. **Note:** when `iemit` is used to generate the memory configuration, the **MemnotWrD0** pin must be used. For further details see *The Transputer Databook 72 TRN 203 01.*

The external memory interface configuration tool `iemit` produces timing diagrams for potential configurations of the memory interface and warns of possible errors in the design. It indicates whether one of the preset configurations that are available would be suitable, or whether it would be necessary to use an externally programmed configuration.

Note: That it is assumed that readers creating memory configuration files are familiar with the memory interface of the processor that they are using. The stages in designing a memory interface, including examples, are described in chapter 2 of *The Transputer Applications Notebook - Systems and Performance.* Further information may also be found in *The Transputer Databook.*

17.2 Running `iemit`

The `iemit` tool can be invoked by the following command line:

▶ `iemit` *options*

where: *options* is a list of one or more options from table 17.1.

> Options are preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.

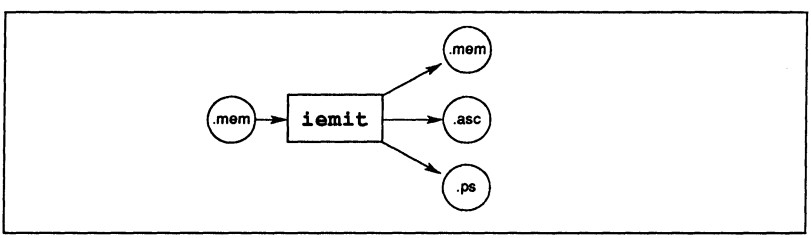
If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
A	Produce ASCII output file.
E	Invoke interactive mode.
F *filename*	Specify input memory configuration file.
I	Select verbose mode. In this mode the user will receive status information about what the tool is doing during operation for example, reading or writing to a file.
O *filename*	Specify output filename.
P	Produce PostScript output file.

Table 17.1 iemit command line options

Note: that if option 'E' is selected i.e. interactive mode, then no other options may be specified on the command line.

The operation of iemit in terms of standard file extensions is shown below:



Examples of use

iemit may be invoked in interactive mode by using one of the following commands:

```
iemit -e                    (UNIX based toolsets)
iemit /e                    (MS-DOS and VMS based toolsets)
```

Output files in ASCII or PostScript may be specified by command options from within interactive mode; alternatively iemit may be invoked in batch mode, to create an output file in one of these formats.

When the tool is invoked in batch mode to produce an output file in either ASCII or PostScript format, then an input file must be supplied using the '**F**' option. It is also mandatory to specify either the '**A**' or '**P**' option. If the '**O**' parameter is not supplied then an output filename will be constructed, from the input filename, with an extension of '**.PS**' for a PostScript output, or '**.ASC**' for an ASCII output.

Example:

The following commands cause **iemit** to produce an output file in PostScript format. The tool is invoked in verbose mode.

UNIX based toolsets:

```
iemit -i -p -f memconfig.mem -o waveform.ps
```

MS-DOS and VMS based toolsets:

```
iemit /i /p /f memconfig.mem /o waveform.ps
```

17.3 Output files

Two different types of output may be produced by **iemit**, these are listed below:

- A memory configuration file suitable for including as an input file to the **ieprom** tool.
- An output file in either ASCII or Postscript format, suitable for inclusion in documentation.

The tool may be used interactively to produce a memory configuration file in text format. This file may then be used as an input file to the **ieprom** tool, thus enabling the memory configuration to be stored on ROM. **iemit** is capable of saving and reloading configurations to allow for design over an extended period and for comparison of different configurations. The memory configuration file is described and an example is given in section 17.7.

Additionally **iemit** may be used to produce an output file which is either a plain ASCII file containing timing data or a file in PostScript format containing waveform diagrams. These formats were chosen so that the results of the program could be easily included in reports or other documentation.

17.4 Interactive operation

When **iemit** is invoked in interactive mode the program will power up with the default standard configuration 31.

The tool's user interface is presented as a number of display pages showing timing data. The displays may be updated by changing the timing parameters, which are accessed from page 1. All inputs are executed immediately so that the user can see the effect on any of the displays. As each page is shown, the user has the option of selecting another page for display by keying in its number. The current configuration may be saved at any time to a specified output file.

The information displayed and options available on each page are described below.

17.4.1 Page 0

This page acts as an index to the others. It shows the title of each page and permits the selection of one of them. An option is provided to enable an input file to initialise the memory configuration. Other options enable the user to selectively generate output files. Options are listed in table 17.2 and an example of the display page is given in figure 17.1.

The user enters an option code followed by the RETURN key. If a file option is specified the user will be prompted for a filename. **Note**: file extensions should be specified, there are no defaults.

17.4.2 Page 1

This page shows the input parameters to **iemit**. It is from these parameters that the tool computes the timing information and the waveforms. Only one parameter may be changed at a time and the display data is immediately updated. An example of the display page is given in figure 17.2.

When the page is displayed, the user has the option to select a new page by entering its number, or entering C to change one of the parameters. In the latter case, a list of parameter identifiers is displayed (see table 17.3) and the user is a prompted to select one. The user may then specify a new value, or by pressing the RETURN key, leave the current selection unchanged. The parameters used for modifying the timing data are described in tables 17.4, 17.5 and 17.6.

Note: that there are two parameters displayed on page 1 which are updated by **iemit** but which cannot be directly updated by the user; they are the EMI clock period Tm and the Wait states (see tables 17.5 and 17.6).

Option	Description
1 to 6	Selects the page to be displayed.
S	Save configuration to a file. The program prompts for the name of a file to which the data will be written, by convention the extension .MEM should be used. Output is a memory configuration file. An error is reported if the data could not be saved. The save file is given comments in its header indicating that it was created by the iemit program.
L	Load previously saved configuration. A filename is prompted for, and the configuration saved in that file is read in and the display data is updated. The program expects a memory configuration file. If loading does not succeed because the file has a bad format, the current configuration is reset to standard configuration 31. If loading fails because the file could not be found or could not be opened for reading, the load is abandoned without losing the current configuration.
A	Output pages in ASCII format to a file. The program prompts for the name of a file to which the data will be written. Output is in plain ASCII format with a form feed (FF) character after each page. It includes full timing information and a representation of the timing diagrams for read and write cycles. An error is reported if the output could not be written.
P	Generate PostScript file. The program prompts for a filename. The program writes to the file a program in the PostScript page description language to draw the timing diagrams for the chosen memory interface configuration. The waveforms shown are the same as those which can be seen by selecting pages 4 and 5. The file produced fully conforms to the PostScript structuring conventions, allowing it to be processed by other programs. The diagram is designed to fit lengthways on an A4 page, and is suitable for inclusion in technical notes and reports. The file can be sent directly to an Apple LaserWriter or other PostScript output device.
Q	Quit - selection of this option exits the program.

Table 17.2 iemit page 0 options

Parameter identifier	Parameter
0 to 6	Page to be displayed
D	Device type
T1	Address setup time before address valid strobe
T2	Address hold time after address valid strobe
T3	Read cycle tristate or write data setup
T4	Extendible data setup time
T5	Read or write data
T6	End tristate or data hold
S0	Nonprogrammable strobe "**notMemS0**"
S1	Programmable strobe "**notMemS1**"
S2	Programmable strobe "**notMemS2**"
S3	Programmable strobe "**notMemS3**"
S4	Programmable strobe "**notMemS4**"
RS	Read cycle strobe name
WS	Write cycle strobe name
R	Refresh period
WM	Write mode
W	Memwait input connection
C	Standard configuration

Table 17.3 **iemit** page 1 parameter identifiers

Parameter	Description
Device type	This parameter enables the program to deduce the time taken for a half cycle of the signal **ProcClockOut**: this is **Tm**, the basic unit of time of the memory interface. A menu of the available devices is displayed and the user is invited to select one: T400-20 T800-17 T414-15 T800-20 T414-17 T800-22 T414-20 T800-25 T425-17 T800-30 T425-20 T800-35 T425-25 T805-25 T425-30 T805-30
Tstates T1-T6	The length of each Tstate T1 to T6, is entered as a number of Tm periods between 1 and 4. (2 Tm periods = 1 clock cycle).
Programmable Strobes S0-S4	The programmed durations of the strobes **notMemS0** to **notMemS4**. The strobes each have two names which can be altered. One which can be up to 9 characters in length, and one consisting of just one character. There should be no embedded spaces in the long names. The short names are used in the timing information on pages 2 and 3, while the long names are used to label the waveforms on pages 4 and 5, and in the PostScript output. The signal names are initialised to sensible defaults. **Note**: that S0 is a fixed strobe, so its duration cannot be changed. The duration of a strobe can be 0 to 31 **Tm** periods. If the value for S1 is set to zero, then **notMemS1** stays high throughout the cycle; if the value for S2, S3 or S4 is set to zero, then the strobe is low for the duration of the cycle.

Table 17.4 **iemit** page 1 parameters

Parameter	Description
`Read strobe name`	The names for the read strobe **notMemRd** can be altered.
`Write strobe name`	The names for the write strobe **notMemWrB** can be altered. Note that because the four byte write strobes have the same timing, only one is considered.
`Refresh period`	The refresh period is given as a number of **ClockIn** periods (18, 36, 54, or 72) or as **Refresh Off** if zero is selected.
`Write mode`	The write mode can be set to **Early** or **Late** to suit the type of memory being used.
`Wait connection`	The **MemWait** input may be connected to one of the strobes S2, S3, S4 by entering 'S2', 'S3' or 'S4' respectively. Alternatively, by specifying a number in the range 1 to 60 **MemWait** may be connected to a simulated external wait state generator. This causes **MemWait** to be held high then to become inactive (low) a set number of **Tm** periods after the start of T2. **Note**: that this mode is not supported directly by the T414; in a final design, a circuit would have to be built to perform this function. If the current connection of **MemWait** causes the signal to become inactive just as **ProcClockOut** is falling during T4, a warning is given that there is a hazard of a **wait race** condition. This is because **MemWait** is sampled on the falling edge of **ProcClockOut** – and if the signal is changing while being sampled, the result is undefined.
`EMI clock period Tm`	The value of **Tm** for a **clockIn** frequency of 5MHz. This is computed from the other parameters and displayed.

Table 17.5 `iemit` page 1 parameters

Parameter	Description
Wait states	The number of wait states in the current configuration. This is computed from the other parameters and displayed.
Standard configuration	The parameters can all be reset to those for one of the built in configurations. There are 13 standard configurations available for the T414, valid configuration numbers being 0 to 11 and 31. For the T400, T425, T800 and the T805 there are 17 standard configurations available, valid configuration numbers being 0 to 15 and 31. If the user selects, for a T414, one of the four configurations which are not available, a message will be displayed indicating that this configuration may not be hardwired on a T414. If the currently set configuration happens to correspond exactly to one of the preset configurations, the tool reports the fact.

Table 17.6 iemit page 1 parameters

17.4.3 Page 2

This page shows general timing information for the interface, such as delays between various strobes and required access times of the memory devices to be used. The user should adjust these figures to allow for delays in external logic.

Table 17.7 lists the timing information displayed on this page while an example of the display is given in figure 17.3.

JEDEC symbol	Parameter description
T0L0L	Cycle time (in both nanoseconds and processor cycles)
TAVQV	Address access time
T0LQV	Access time from **notMemS0**
TrLQV	Access time from **notMemRd**
TAV0L	Address setup time
T0LAX	Address hold time
TrHQX	Read data hold time
TrHQZ	Read data turn off
T0L0H	**notMemS0** pulse width low
T0H0L	**notMemS0** pulse width high
TrLrH	**notMemRd** pulse width low
TrL0H	Effective **notMemRd** width
T0LwL	**notMemS0** to **notMemWrB** delay
TDVwL	Write data setup time
TwLDX	Write data hold time 1
TwHDX	Write data hold time 2
TwLwH	Write pulse width
TwL0H	Effective **notMemWrB** width

Table 17.7 General timing parameters

The total cycle time is given in nanoseconds and in processor clock cycles. The only option available from this page is to select another page for display.

17.4.4 Page 3

This page gives timing information of special interest to designers working with dynamic memory, including various access times and the time for 256 refresh cycles. With this information the designer can ensure that the requirements of the memory devices to be used are met. The user should adjust these figures to allow for delays in external logic. Table 17.8 lists the DRAM timing parameters.

JEDEC symbol	Parameter description
T1L1H	**notMemS1** pulse width
T1H1L	**notMemS1** precharge time
T3L3H	**notMemS3** pulse width
T3H3L	**notMemS3** precharge time
T1L2L	**notMemS1** to **notMemS2** delay
T2L3L	**notMemS2** to **notMemS3** delay
T1L3L	**notMemS1** to **notMemS3** delay
T1LQV	Access time from **notMemS1**
T2LQV	Access time from **notMemS2**
T3LQV	Access time from **notMemS3**
T3L1H	**notMemS1** hold (from **notMemS3**)
T1L3H	**notMemS3** hold (from **notMemS1**)
TwL3H	**notMemWrB** to **notMemS3** lead time
TwL1H	**notMemWrB** to **notMemS1** lead time
T1LwH	**notMemWrB** hold (from **notMemS1**)
T1LDX	Write data hold from **notMemS1**
T3HQZ	Read data turn off
TRFSH	Time for 256 refresh cycles (in microseconds)

Table 17.8 DRAM timing parameters

The only option available from this page is to select another page for display. An example of the display is given in figure 17.4.

17.4.5 Page 4

This page shows graphically the timing for a memory read cycle. An example of the display page is given in figure 17.5.

The **Tstates** and strobes are labelled, and bus activity is shown. The point where data are latched into the processor is also indicated.

At the top of the page is displayed the processor clock and the Tstates, a number indicating the Tstate, 'W' indicating a wait state, and 'E' indicating a state that is inserted to ensure that T1 starts on a rising edge of the processor clock.

Below this are displayed the waveforms of the programmable strobes and the read, write and address/data strobes. Each of these strobes is labelled with the corresponding label parameter.

The point at which the read data is latched is indicated by a '^' beneath the read cycle address/data strobe.

The **MemWait** waveform shows the input to the **MemWait** pin. If the wait input is a number then it goes low *n* Tm periods after the end of T1 and high again at the end of T6, if the wait input is connected to a strobe it goes low and then high when that strobe does so.

If the cycle is too long to fit horizontally on the screen, it may be scrolled left and right using the `L` and `R` options. The displayed area moves by about 15 characters each time these options are used.

17.4.6 Page 5

Page 5 shows the waveforms for a memory write cycle. The display is similar to that of page 4, indeed the read and write cycle diagrams are combined when the PostScript output is produced.

Scrolling the display to the left or right is permitted in the same way as for page 4.

An example of the display page is given in figure 17.6.

17.4.7 Page 6

This page gives a **configuration table** for the current configuration. This is a listing of the data which have to be placed in a ROM situated at the top of the transputer's memory map in order to achieve the desired configuration. The table consists of 36 words of data, but only the least significant bit in each is used. The address and contents are given for each location. **Note**: when **iemit** is used to generate the memory configuration, the **Memconfig** pin must be connected to **MemnotWrD0**.

An example of the display page is given in figure 17.7.

Note: that if page 1 indicates that the configuration is one of the transputer's preset ones, there will be no need for a ROM; configuration can be achieved by connecting the **MemConfig** pin of the device to one of the address/data lines.

17.5 Example iemit display pages

```
Page 0      T414/T800 External Memory Interface Program
            ===========================================

            Page 0: Index - this page
                 1: EMI configuration parameters
                 2: General timing
                 3: Dynamic RAM timing
                 4: Read cycle waveforms
                 5: Write cycle waveforms
                 6: Configuration table

Please enter 1...6 to see a new page;
             <S>   to save configuration to a file;
             <L>   to load a saved configuration;
             <A>   to generate an ASCII listing of all pages to a file;
             <P>   to generate PostScript file of waveforms;
             <Q>   to exit the program
:
```

Figure 17.1 Example **iemit** display page 0

```
Page 1          EMI  Configuration  Parameters
                ================================

Device type                              T414-20
EMI clock period (Tm)                    25 ns at ClockIn
                                                 =  5MHz
Wait States                              0
Address setup time                       T1:  1 periods Tm
Address hold time                        T2:  1 periods Tm
Read cycle tristate/Write data setup     T3:  1 periods Tm
Extended for wait                        T4:  1 periods Tm
Read or write data                       T5:  1 periods Tm
End tristate / Data hold                 T6:  1 periods Tm
Non-Programmable strobe "notMemS0 " "0" S0
Programmable strobe "notMemS1 " "1" S1:  30 periods Tm
Programmable strobe "notMemS2 " "2" S2:   1 periods Tm
Programmable strobe "notMemS3 " "3" S3:   3 periods Tm
Programmable strobe "notMemS4 " "4" S4:   5 periods Tm
Read  cycle strobe  "notMemRd " "r"
Write cycle strobe  "notMemWrB" "w"
Refresh period  72  clockin periods              Wait   0
Write mode      Late                     Configuration   0
```

Figure 17.2 Example **iemit** display page 1

```
Page 2                  General Times
                        =============

Symbol       Parameter          min(ns) max(ns)  notes

T0L0L   Cycle time               150       -  = 3 processor cycles
TAVQV   Address access time        -     125
T0LQV   Access time from 0         -     100
TrLQV   Access time from r         -      50
TAV0L   Address setup time        25       -
T0LAX   Address hold time         25       -
TrHQX   Read data hold time        0       -
TrHQZ   Read data turn off         -      25
T0L0H   0 pulse width low        100       -
T0H0L   0 pulse width high        50       -
TrLrH   r pulse width low         50       -
TrL0H   Effective r width         50       -
T0LwL   0 to w delay              50       -
TDVwL   Write data setup time     25       -
TwLDX   Write data hold time 1    75       -
TwHDX   Write data hold time 2    25       -
TwLwH   Write pulse width         50       -
TwL0H   Effective w width         50       -
```

Figure 17.3 Example **iemit** display page 2

```
Page 3                Dram Times
                      ==========

Symbol       Parameter          min(ns)  max(ns) notes

T1L1H   1 pulse width             125       -
T1H1L   1 precharge time           25       -
T3L3H   3 pulse width              25       -
T3H3L   3 precharge time          125       -
T1L2L   1 to 2 delay               25       -
T2L3L   2 to 3 delay               50       -
T1L3L   1 to 3 delay               75      75
T1LQV   Access time from 1          -     100
T2LQV   Access time from 2          -      75
T3LQV   Access time from 3          -      25
T3L1H   1 hold (from 3)            50       -
T1L3H   3 hold (from 1)           100       -
TwL3H   w to 3 lead time           50       -
TwL1H   w to 1 lead time           75       -
T1LwH   w hold (from 1)           100       -
T1LDX   Wr data hold from 1       125       -
T3HQZ   Read data turn off          -      25
TRFSH   256 refresh cycles          -    3650 Time is in microseconds
```

Figure 17.4 Example **iemit** display page 3

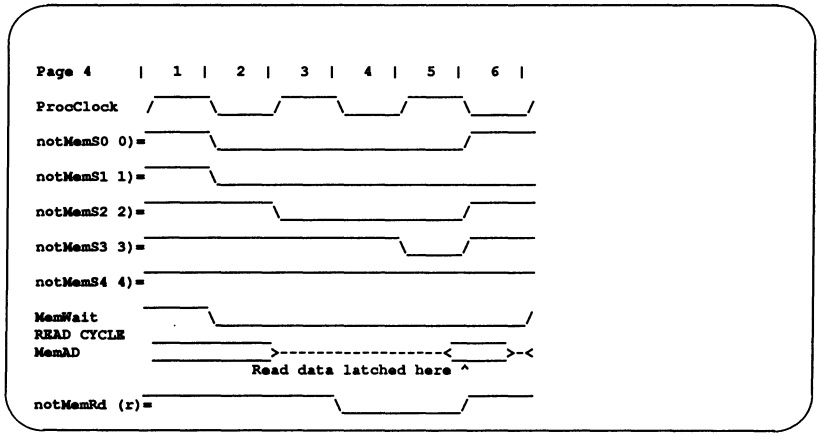


Figure 17.5 Example **iemit** display page 4

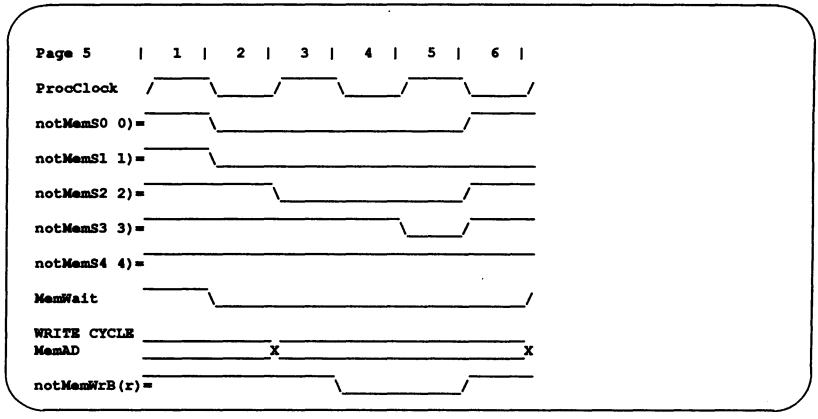


Figure 17.6 Example **iemit** display page 5

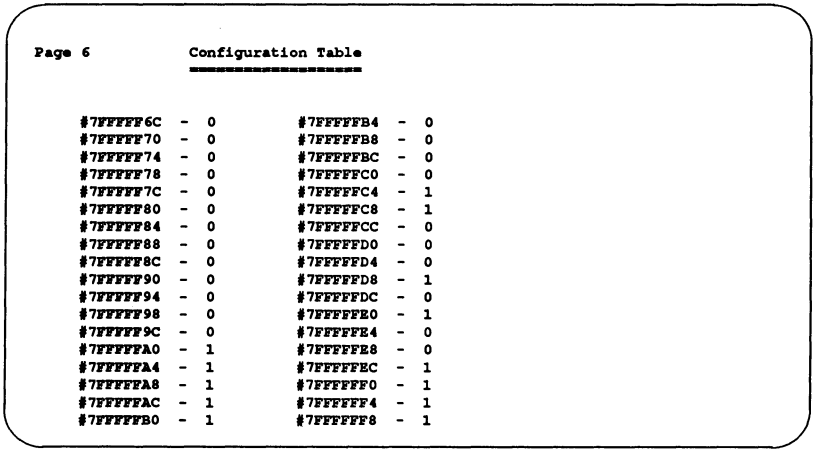


Figure 17.7 Example **iemit** display page 6

17.6 iemit error and warning messages

The following is a list of error and warning messages the tool can produce:

Wait race

If one of the programmable strobes is used to extend the memory cycle then the strobe must be taken low an even number of periods Tm after the start of the memory interface cycle. If the strobe is taken low an odd number of periods after the start then a wait race warning will appear. Should this warning appear, it will remain on display on page 1, until the race condition is removed. Further information can be obtained from reference 1, listed at the start of this chapter.

Input out of range

If the value entered for a numeric parameter is outside the range valid for that parameter, an input out of range warning is displayed, the value cleared from the screen and the program waits for a new value.

MemWait connection error

If an attempt is made to connect S1 to the **MemWait** input an error is displayed because it is a meaningless operation.

Configuration cannot be hardwired on a T414

The transputers which have a configurable memory interface all have (with the exception of the T414) 17 standard memory configurations available to them. The T414 only has a choice of 13 standard configurations. If the standard configurations 12, 13, 14 or 15 are selected for a T414 the above warning message will be displayed against the selection on page 1.

Unable to open configuration file '*filename*'

This can occur when attempting to load a memory configuration file and indicates that the tool cannot find the specified input file. Check the spelling of the filename and/or that the file is present.

Command line parsing error

An option has been specified that the tool does not recognise.

No input file specified

This indicates that when trying to invoke the tool to produce an output file, the user has not specified a memory configuration file to use as input.

One and only one of options A or P must be specified

This indicates that when trying to produce an output file, the user has not specified whether the output is to be in ASCII or PostScript format.

Unable to open output file *'filename'*

An output filename has been specified incorrectly. Check the format of the filename.

17.7 Memory configuration file

Memory configuration files are text files which may be generated by a standard text editor or by using the memory interface configuration tool **iemit**, see section 17.2.

If the user has existing memory configuration files created by **iemi** (a previous version of **iemit**) then the user will need to convert them from the old file format to the file format used by the current EPROM tools. This is achieved by using the memory configuration conversion tool **icvemit**, see section 17.8.

By convention memory configuration files have the file extension **.mem**. The file consists of a sequence of statements and comments. The following are considered to be comments:

- Blank lines
- Any line whose first significant characters are '--'
- Any portion of a line following '--'.

Comments are ignored by the **ieprom** and **iemit** tools. Statements are all other lines within the file; they may be interspersed with comments.

Individual statements are constructed of the statement and an associated parameter. These must be separated by at least one space or tab but extra spaces may be inserted before, between, or after them for aesthetic purposes.

The statements defined are listed along with their parameters in table 17.9. Further information about specifying parameters is given in section 17.4.2.

Statement	Parameters
`standard.configuration`	0 to 13 or 31 for T414 processors. 0 to 15 or 31 for T400, T425, T800 and T805 processors.

`device.type`	One of the following devices:
	
	`T400-20` `T800-17`
	`T414-15` `T800-20`
	`T414-17` `T800-22`
	`T414-20` `T800-25`
	`T425-17` `T800-30`
	`T425-20` `T800-35`
	`T425-25` `T800-25`
	`T425-30` `T805-30`

`t1.duration`, `t2.duration`, `t3.duration`, `t4.duration`, `t5.duration`, `t6.duration`	1 to 4 Tm periods. (2 Tm periods = 1 clock cycle). Defines the length in Tm periods of Tstates, T1 to T6, of the memory cycle.
`s0.label`, `s1.label`, `s2.label`, `s3.label`, `s4.label`	Each of these parameters accepts two text strings. They are the long (up to 9 characters) and short (1 character) names of the strobes **notMemS0** to **notMemS4**. The names should not contain embedded spaces. Names longer than the permitted number of characters will be truncated.
`rs.label`	As above, the long and short names for the read strobe **notMemRd**.
`ws.label`	As above, the long and short names for the read strobe **notMemWrB**.
`s1.duration`	0 to 31 Tm periods. The S1 strobe goes low at the start of Tstate 2. This parameters defines the number of Tm periods before it goes high.
`s2.duration`, `s3.duration`, `s4.duration`	0 to 31 Tm periods. The S2 to S4 strobes all go high at the end of Tstate 5. These parameters define the number of Tm periods before each strobe goes low.

Table 17.9 Memory Configuration file statements

Statement	Parameters
`refresh.period`	18, 36, 54, 72 or the string "Disabled". This parameter defines the period between refresh cycles as a count of **ClockIn** cycles.
`write.mode`	String value either: "Early" or "Late". Defines the write mode.
`wait.connection`	S2, S3, S4 or a value in the range 0 to 60. This parameter connects **MemWait** to one of the strobes S2, S3, S4 or to simulated external wait state generator.

Table 17.10 Memory Configuration file statements

Example memory configuration file

```
--        Memory interface configuration for
--        build xxx of processor board.

device.type          := T800-25
t1.duration          := 3  -- Take 3 state to setup
                           -- address.
t2.duration          := 2
t3.duration          := 1
t4.duration          := 2
t5.duration          := 1
t6.duration          := 1
s1.duration          := 5
s2.duration          := 1
s3.duration          := 2
s4.duration          := 9
s0.label             := ALE 0
s1.label             := RAS 1
s2.label             := MUX
s3.label             := CAS
s4.label             := WAIT
rs.label             := notMemRd
ws.label             := notMemWrB
refresh.period       := 36
write.mode           := EARLY
wait.connection      := S4
```

17.8 Memory interface conversion tool icvemit

This tool is provided to convert memory configuration files produced by **iemi** (a previous version of **iemit**) to the file format recognised by the current release of **iemit** and **ieprom**.

The tool will take, as input, the 'save' file produced by **iemi** and convert it to a memory configuration file in a format which may be read by the current release of the EPROM tools.

17.9 Running icvemit

The **icvemit** tool can be invoked by the following command line:

▶ **icvemit** *filename* {*options*}

where: *filename* is the input file; this file must have been created by the tool **iemi** released with previous versions of the transputer toolset.

options is a list of one or more options from table 17.11.

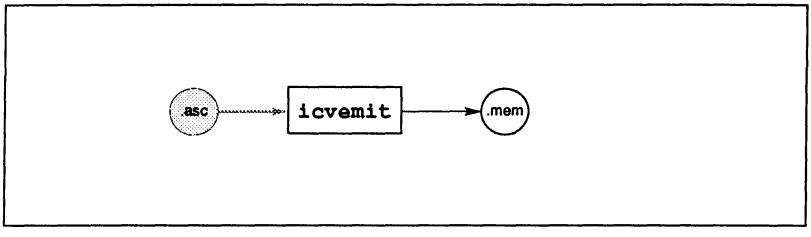
Options are preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
Options may be entered in upper or lower case and can be given in any order.
Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
I	Select verbose mode. In this mode the user will receive status information about what the tool is doing during operation eg. reading or writing to a file.
O *filename*	Specify output filename. Saves the output to a specified filename. If the option is not supplied then the output will be placed in a file with the same name as the input file but with the extension of "mem".

Table 17.11 **icvemit** options

The operation of **icvemit** in terms of standard file extensions is shown below:



Note: the file extension of the input file pertains to previous issues of the toolset.

Example

`icvemit memconfig.asc /i /o memconfig.mem`	MS-DOS and VMS based toolsets.
`icvemit memconfig.asc -i -o memconfig.mem`	UNIX based toolsets.

17.10 icvemit error messages

The following is a list of error and warning messages the tool can produce:

Unable to open configuration file '*filename*'

Indicates that the tool cannot find the specified input file. Check the spelling of the filename and/or that the file is present.

Command line parsing error

This indicates that an option has been specified that the tool does not recognise.

No input file specified

This indicates that when trying to invoke the tool to produce an output file, the user has not specified a memory configuration file to use as input.

Unable to open output file *'filename'*

An output filename has been specified incorrectly. Check the format of the filename.

18 `ieprom` – EPROM program convertor

This chapter describes the EPROM Hex tool **`ieprom`**. This tool is used to convert a ROM-bootable file into one or more files suitable for blowing into an EPROM.

The chapter describes how to invoke **`ieprom`** and gives details of the command line syntax. It describes the control file which the tool accepts as input and provides background information on the layout of the code in the EPROM. A description of the various file formats which may be output by the tool is given, including block mode where the output is split up over a number of files. The chapter ends with a list of error messages which may be generated by the tool.

18.1 Introduction

The INMOS EPROM software is designed so that programs which have been developed and tested using the INMOS toolset may be placed in ROM with only minor modification (see below).

This has the advantages that an application need not be committed to ROM until it is fully debugged and the actual production of the ROMs can be done relatively late in the development cycle without the fear of introducing new problems.

If a network of transputers is being used, only the root transputer needs to be booted from ROM; once this has been booted it will boot its neighbours by link.

Figure 18.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

Some 32 bit transputers have a configurable external memory interface. For these transputers a memory configuration file may be created and blown into ROM together with the application. A description of memory configuration files and how to create them is given in chapter 17.

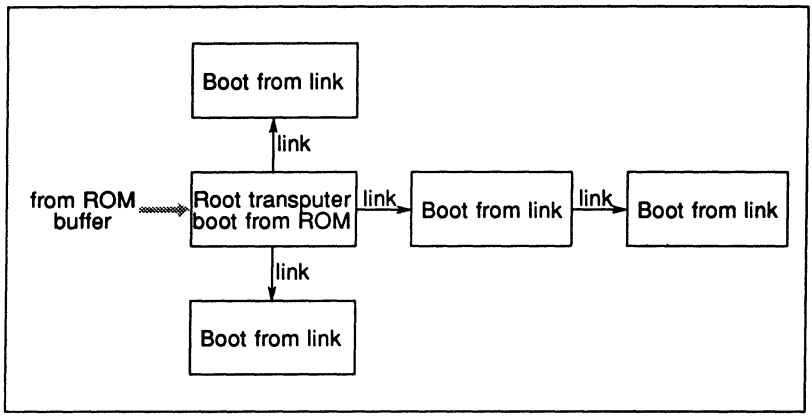


Figure 18.1 Loading a network from ROM

18.2 Prerequisites to using the hex tool ieprom

For an application file to be suitably formatted for blowing into ROM it must have been configured to be booted from ROM rather than booted from link. This selection is made by specifying the appropriate command line option when using the **icconf** and **icollect** tools. See chapters 12 and 13 respectively. It is also essential that all C programs, including those targeted at a single processor are configured using **icconf**. C programs prepared with the **icollect** '**T**' option do not have the required format to be suitable input for **ieprom**.

18.3 Running ieprom

ieprom takes as input a control file and outputs one or more files which may be blown into ROM by an EPROM programmer.

The control file, in text format, specifies the root transputer type, the name of the bootable file containing the application, the memory configuration file (if one is being used), the amount of space required on the EPROM and the format that the output is to take. Available output formats are: binary, hex dump, Intel, Extended Intel or Motorola S-Record format.

The **ieprom** tool is invoked by the following command line:

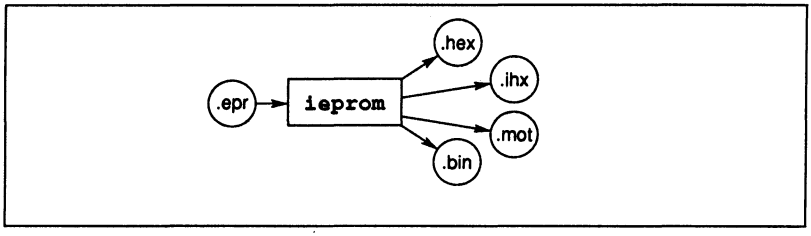
▶ **ieprom** *filename* {*option*}

where: *filename* is the name of the control file.

option may take the value **I** which selects verbose mode. In this mode the user will receive status information about what the tool is doing during operation for example reading or writing to a file. If option '**I**' is specified it must be preceded by '-' for UNIX based tools or '/' for MS-DOS and VMS based tools.

If no arguments are given on the command line a help page is displayed giving the command syntax.

The operation of **ieprom** in terms of standard file extensions is shown below.



18.3.1 Examples of use

ieprom may be invoked in verbose mode by using one of the following commands:

```
ieprom -i mycontrol.epr          (UNIX based toolsets)

ieprom /i mycontrol.epr    (MS-DOS and VMS based toolsets)
```

18.4 ieprom control file

The control file is a standard text file, prepared by an editor; it consists of comments and statements.

The following are considered to be comments:

- Blank lines
- Any line whose first significant characters are '--'
- Any portion of a line following '--'.

Comments are ignored by the **ieprom** tool.

Statements are all other lines within the file. They may be in any order, except that the four statements defining a block must immediately follow the statement **output.block** (see table 18.2). Statements may be interspersed with comments.

Individual statements are constructed of the statement and an associated parameter. These must be separated by at least one space or tab but extra spaces may be inserted before, between, or after them for aesthetic purposes. The statements defined are listed along with their parameters in tables 18.1 and 18.2.

Statement	Parameter/Description
`root.processor.type`	T2, T4 or T8 This statement has a keyword as its parameter. It specifies the root processor type as being T2 (16 bit processor), T4 (32 bit processor), or T8 (32 bit processor with a floating point unit). This statement must be present in the control file.
`bootable.file`	*filename* This statement specifies the file that contains the output of `icollect`, usually the application plus its ROM loader(s). This file is inserted into the EPROM with the comment bootstrap at its head removed. This statement must be present in the control file.
`memory.configuration`	*filename* This statement specifies a memory configuration file to be included in the EPROM image. This file is a standard memory configuration description (see chapter 17 for details). If this statement is absent from the control file then no memory configuration will be inserted in the image.
`eprom.space`	hexadecimal number This statement specifies the size of the EPROM space in bytes. The space may actually contain several physical devices. This statement must be present in the control file.
`output.format`	`binary`, `hex`, `intel`, `extintel` or `srecord` This statement takes a keyword as a parameter. It specifies the type of the records going to the output file, as binary output, a plain hex dump, Intel format, Extended Intel format, or Motorola S-Record format respectively. If this statement is absent from the control file then the output will be a simple hex dump.

Table 18.1 `ieprom` control file statements

Statement	Parameter/Description
output.all **output.block**	*filename* *filename* These two statements specify the output file. By convention the file extension **.epr** should be used. **output.all** means that all of the image is to be output to one file. **output.block** specifies that a block of data is to be output to the specified file. It must be followed by the four statements that define that block; these are detailed next. The control file must contain one **output.all** statement, or one or more **output.block** statements.
start.offset	hexadecimal number This statement specifies the offset, into the EPROM space, of the start of a block. One of these statements must follow each **output.block** statement.
end.offset	hexadecimal number This statement specifies the offset, into the EPROM space, of the end of a block. One of these statements must follow each **output.block** statement.
byte.select	decimal number or **all** This statement takes a decimal number, or the keyword **all**, as a parameter. It specifies which bytes in a word are to be output in this block. The number takes values 0, 1, 2 or 3 for 32 bit processors, and 0 or 1 for 16 bit processors. One of these statements must always follow each **output.block** statement.
output.address	hexadecimal number This statement specifies the address in the EPROM programmer's memory, at which the block is to be output. For **output.all** the output address is always zero. One of these statements must always follow each **output.block** statement.

Table 18.2 **ieprom** control file statements

Example control file

```
-- -----
--
--        EPROM description file for
--        build of complicated example
--
-- -----

root.processor.type  T4

bootable.file        wiggle.btr
memory.configuration slowacc.mem
eprom.space          20000

output.format        SRECORD

output.block         part1.mot
  start.offset       00000
  end.offset         0FFFF
  byte.select        0
  output.address     00000

output.block         part2.mot
  start.offset       00000
  end.offset         0FFFF
  byte.select        1
  output.address     00000

output.block         part3.mot
  start.offset       00000
  end.offset         0FFFF
  byte.select        2
  output.address     00000

output.block         part4.mot
  start.offset       00000
  end.offset         0FFFF
  byte.select        3
  output.address     00000

          etc ...
```

18.5 What goes in the EPROM

This section describes the contents of the EPROM, the reasons behind the code layout and the function of those components inserted by **ieprom**.

The content of the EPROM when blown includes the bootable file, traceback data and jump instructions to enable the processor to find the start of the bootable file. Should the user define the memory configuration this information will also be placed in the EPROM. The general layout of the code in the EPROM is shown in figure 18.2.

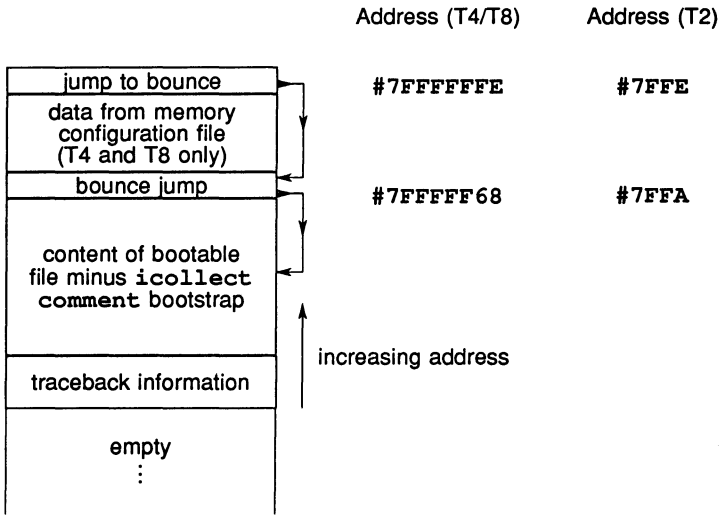


Figure 18.2 Layout of code in EPROM

18.5.1 Memory configuration data

Memory configuration data, when present, is placed immediately below the top word of the EPROM. The top word holds the first instructions to be executed if the transputer is booting from ROM.

If the processor has a configurable memory interface it will scan the memory configuration data held on the EPROM, before executing the first instructions. If a standard memory configuration is being used there should be no memory configuration data present and the processor will ignore this section of the EPROM.

18.5.2 Jump Instructions

The first instruction executed by the processor when booting from EPROM, is located at (MOSTPOS INT) - 1: this is **#7FFFFFFE** for 32-bit machines and **#7FFE** for 16-bit machines. The first two instructions cause a backwards jump to be made, with a distance of up to 256 bytes; however, since most applications are larger than 256 bytes it is necessary for **ieprom** to insert a **bounce** jump to the start of the bootable file.

18.5.3 Bootable file

The bootable file will have been produced by the collector tool **icollect**, using a boot from ROM loader. The **comment** bootstrap, containing traceback information, originally added to this file by **icollect**, is stripped off by **ieprom**.

The bootable file is placed in the EPROM such that the start of the file is placed at the lowest address, with the rest of the file being loaded in increasing address locations. The end of the file is placed immediately below the **bounce** jump instruction, which points to the start of the bootable file.

18.5.4 Traceback Information

ieprom creates its own traceback information consisting of the name of the control file and the time at which **ieprom** ran. This information is placed below the start of the bootable file. **Note** at present this information is not used by any of the tools.

18.6 ieprom output files

The tool can produce output in a form readable by the user or in a form readable by EPROM programming devices. The following formats are supported:

Binary output

Hex dump

Intel hex format

Intel extended hex format

Motorola S-record format

Whichever form is used, it is sometimes necessary to output the data in blocks. Block mode operation is discussed in section 18.7.

Note: there is no output for unused areas of the EPROM. If the buffer in the EPROM programmer is not initialised before loading the files produced by this program into it, unused areas of the EPROM will be filled with random data. Although the operation of the bootstrap code and loader programs will not be affected by the presence of random data, these areas of the EPROM cannot subsequently be programmed without erasing the whole device.

18.6.1 Binary output

This file is in binary format and simply contains all bytes output. There is no additional information such as address or checksums.

18.6.2 Hex dump

This simple format is intended to be used to check the output from the program. The dump consists of rows of 16 bytes each, prefixed by the address in the initial byte of each row. The format contains no characters other than the hexadecimal digits, the space character and newlines.

18.6.3 Intel hex format

This is a commonly used protocol for EPROM programming equipment. A sequence of **data records** is sent. Each record contains a few bytes of information, a start address and a checksum. In addition, a special record marks the end of a transmission. Since the format only supports 16-bit addresses, any longer addresses will generate an error message. Records produced by this program contain at most 32 bytes each.

18.6.4 Intel extended hex format

This format, also known as Intel 86 format, is similar to Intel hex, but adds another type of record. The new **type 02 record** is used to specify addresses of more than 16 bits. The type 02 record contains a 16-bit field giving a **segment base offset**. This value is shifted left four places and added to subsequent addresses. This mimics the operation of the segment registers on the Intel 8086 range of microprocessors. The segment base offset value persists until the next type 02 record occurs. This format therefore allows addresses up to 20 bits in length. Again, longer addresses will generate an error message. The program minimises the number of type 02 records inserted in its output.

18.6.5 Motorola S-record format

This format is another well known industry standard; it consists of a header record, data records, and finally an image end record. The advantage of this format is that, by the use of different data record types, it can support 16, 24, or 32 bit addresses. This program uses whichever data record type is necessary.

18.7 Block mode

Block mode is a term used to describe the output from **ieprom**, when more than one output file is produced. The user defines how the data is to be split between files using control file statements. (See table 18.2).

18.7.1 Memory organisation

In order to understand the ideas behind block mode operation it is helpful to understand the way memory is organised in a 16 or 32 bit transputer.

In general, a transputer with a 32 bit data bus will expect to read from memory in 32 bit words; the addresses of these words will be on word boundaries (i.e. the address will always be divisible by 4, the two least significant bits will be 0). EPROM devices, however, are usually 8 bits wide, and so it is necessary to have 4 EPROMs side by side to make up the 32 bit width. We identify these 4 devices as being byte **n** (**n** = 0, 1, 2 or 3), where the least two significant bits of the address would together have the value **n**.

Similarly a 16 bit transputer will expect to read from memory in 16 bit words. The address of each word will always be divisible by 2. The two EPROM devices required to make up the 16 bit width will be identified as bytes (**n** = 0 or 1).

18.7.2 When to use block mode

Block mode has three uses:

- When the EPROM programmer being used is unable to split up the bytes from its input, in order to program separate byte wide devices.
- When the EPROM programmer has insufficient memory to hold the entire image.
- When it is required for some reason, to load the program to a different address in the EPROM programmer to that which it will occupy in the EPROM space.

18.7.3 How to use block mode

When block mode is to be used, the user must first decide on the blocks to be output. For each block required an **output.block** statement must be specified in the control file. Each **output.block** statement must be followed by the four statements:

```
start.offset

end.offset

byte.select

output.address
```

ieprom will scan the entire image and output those bytes that have an eprom space address between **start.offset** and **end.offset** and whose byte address matches the **byte.select** value. It will output this data to contiguous addresses starting at **output.address**.

Note: if the image does not occupy all of the EPROM space then there may be some space at **output.address** before the data starts.

18.8 Example control files

Simple output

For this example we will assume that the application is in **bootable.btr**, there is no memory configuration, there is 128k of EPROM space, and the programmer can take the whole image in one file.

Then the control file will look like :-

```
--        EPROM description file for
--        build of network program

root.processor.type  T4

bootable.file        bootable.btr
eprom.space          20000
output.format        srecord

output.all           image.mot
```

Using block mode

For this example we will assume that the application is in **embedded.btr**, there is a memory configuration in **fastsram.mem**, there is 16k of EPROM, the image is to be split into four blocks of 4k EPROMS, and that these EPROMS are to be programmed from locations 0000, 1000, 2000, and 3000 in the EPROM programmer's memory.

The control file will look like :-

```
--          EPROM description file for
--          build of embedded system

root.processor.type   T8

bootable.file         embedded.btr
memory.configuration  fastsram.mem

eprom.space           4000

output.format         intel

output.block          part1.ihx
  start.offset        0000
  end.offset          3FFF
  byte.select         0
  output.address      0000

output.block          part2.ihx
  start.offset        0000
  end.offset          3FFF
  byte.select         1
  output.address      1000

output.block          part3.ihx
  start.offset        0000
  end.offset          3FFF
  byte.select         2
  output.address      2000

output.block          part4.ihx
  start.offset        0000
  end.offset          3FFF
  byte.select         3
  output.address      3000
```

18.9 Error and warning messages

The following is a list of error and warning messages the tool can produce:

Command line parsing error

This indicates that a command line option has been specified that the tool does not recognise.

No input file specified

This indicates that when trying to invoke the tool the user has not specified a control file to use as input.

Unable to open control file '*filename*'

The control file specified cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open configuration file '*filename*'

The memory configuration file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open bootable file '*filename*'

The bootable file specified in the control file cannot be found. Check the spelling of the filename and/or that the file is present.

Unable to open output file *'filename'*

An output filename has been specified incorrectly. Check the format of the filename.

Control file error

This message will be received whenever an error is found in the format of the control file. A self explanatory message will be appended, giving details of what the tool expects the format to be.

19 ilibr – librarian

This chapter describes the librarian tool **ilibr** that integrates a group of compiled code files into a single unit that can be referenced by a program. The chapter begins by describing the command line syntax, goes on to describe some aspects of toolset libraries, followed by some hints about how to build efficient libraries from separate modules. The chapter ends with a list of error messages which may be generated by the tool.

19.1 Introduction

The librarian builds libraries from one or more separately compiled units supplied as input files. The input files may be any of the following:

- Object code files produced by **icc**.
- Library files already generated by **ilibr**.
- Object code files produced by the convertor tool **icvlink**.
- Object code files produced by other compatible INMOS compilers such as **oc** the TCOFF occam 2 compiler.

The library, once built, will contain an index followed by the concatenated modules. The index is generated and sorted by the librarian to facilitate rapid access of the library content by the other tools in the toolset, for example, the linker.

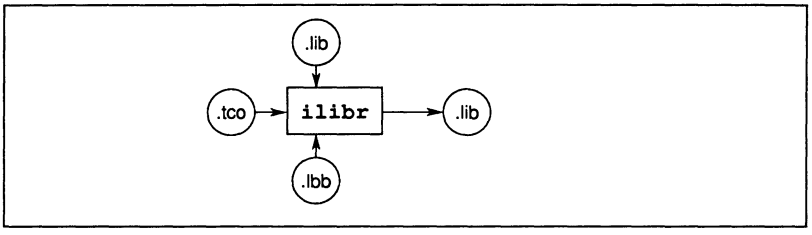
19.2 Running the librarian

The librarian takes a list of compiled files in TCOFF format and integrates them into a single object file that can be used by a program or program module. Each module in the input list becomes a selectively loadable module in the library. Input files can either be specified as a list on the command line or in *indirect files*.

Compiled files may be concatenated for convenience before using the librarian. This may prove useful when dealing with a large number of input files. The number of file names allowed on a command line is system dependent. To avoid overflow, files may be concatenated or an indirect file used. It is the user's responsibility to ensure that the concatenation process does not corrupt the modules, for example by omitting to specify that the concatenation is to be done in binary mode.

Note: when a library file is used as a component of a new library, its index is discarded by **ilibr**.

The operation of the librarian in terms of standard file extensions is shown below.



To invoke the librarian use the following command line:

▶ **ilibr** [*filenames*] {*options*}

where: *filenames* is a list of input files or indirect files in any combination separated by spaces. Any string not recognised as an option is treated as a filename.

options is a list of one or more options, in any order, from Table 19.1.

> Options must be preceded by '**-**' for UNIX based toolsets and '**/**' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.
>
> Options must not appear within indirect files.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Example of use

```
ilibr myprog.t4x myprog.t8x
```

In this example, the files **myprog.t4x** and **myprog.t8x** (compiled for T4 and T8 transputers respectively) are used to create a library. Because no output file name is specified on the command line, the library will be given the name **myprog.lib**.

Option	Description
F *filename*	Specifies a library indirect file.
I	Displays progress information as the library is built.
L	Loads the librarian onto a transputer board and terminates.
O *filename*	Specifies an output file. If no output file is specified the name is taken from the first input file and a `.lib` extension is added.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

Table 19.1 `ilibr` command line options

19.2.1 Default command line

A set of default command line options can be defined for the tool using the ILIBRARG environmental variable. Options must be specified using the syntax required by the command line.

19.2.2 Library indirect files

Library indirect files are text files that contain lists of input files, directives to the librarian, and comments. Filenames and directives must appear on different lines. Comments must be preceded by the double dash character sequence '`--`', which causes the rest of the line to be ignored. By convention indirect files are given the `.lbb` extension.

Indirect files may be nested within each other, to any level. This is achieved by using the `#INCLUDE` directive. By convention nested indirect files are also given the extension `.lbb`.

The following is an example of an indirect file:

```
-- user's .lbb file

userproc1.tco              -- single modules
userproc2.tco
userproc3.tco
myconcat.tco               -- concatenation of modules
#INCLUDE indirect.lbb      -- another indirect file
```

```
userproc4.tco                -- another single module
```

The contents of a nested indirect file will effectively be expanded at the position it occured.

To specify indirect files on the command line each indirect filename must be preceded by the '**F**' option.

19.3 Library modules

Libraries are made up of one or more selectively loadable modules. Each module is selected via the library index. A module is the smallest unit of a library that can be loaded separately.

19.3.1 Selective loading

Libraries can contain the same routines compiled for different transputer types and (if non-C code is used) in different error modes. The linker decides which library modules are used and selects the library modules best suited to the compilation units.

19.4 Library usage files

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They consist of a list of separately compiled units or libraries referenced within a particular library. The **.liu** files required by the toolset's libraries are supplied by INMOS. Library usage files are text files and can be edited.

If the **imakef** tool is used then library usage files should be created for all libraries that are supplied without source. This is to enable the **imakef** tool to generate the necessary commands for linking. Library usage files can be created for a specific library by invoking **imakef** and specifying a **.liu** target.

Such files are given the same name as the library file to which they relate but with an **.liu** extension.

19.5 Building libraries

This section describes the rules that govern the construction of libraries and contains some hints for building libraries.

19.5.1 Rules for constructing libraries

1 Routines of the same name in a library must be compiled for different transputer types and error modes.

2 Libraries that contain modules compiled for a transputer class (i.e. TA or TB) are treated as though they contain a copy for each member of the class.

3 Libraries that contain modules compiled in UNIVERSAL mode are treated as though they contain a copy for each of the three mixed language programming error modes (HALT, STOP, and UNIVERSAL).

4 Libraries that contain modules compiled with interactive debugging information enabled are treated as though they also contain a copy of the modules with interactive debugging disabled.

19.5.2 Hints for building libraries

Routines that are likely to be used together in a program or procedure (such as routines for accessing the file system) can be incorporated into the same library. At a lower level, routines that will *always* be used together (such as those for opening and closing files) can be incorporated into the same module.

Libraries can contain the same routines compiled for different transputer types, in different error modes (for mixed language programs only), and with interactive debugging information enabled or disabled. Only those modules actually used by the program are incorporated by the compiler and linked in by the linker.

Where possible compile library input files with debugging enabled. This enables the debugger to locate the library source if an error occurs inside the library.

19.5.3 Optimising libraries

The librarian creates a library index which is used by the linker to select the required modules. The librarian sorts the index so that for a given processor type the optimum module is always selected by the linker.

It is possible, by compiling for transputer classes, to optimise the size and content of any libraries to improve the speed of code execution and to provide the best code for a given processor. Transputer types and classes are described in section 5.3.

The following three approaches to building a library may be followed as appropriate; the third approach provides the greatest level of optimisation.

Library build targeted at specific transputer types

This method of building a library will limit the use of the library modules to specific transputer types. It is recommended as the simplest strategy to use provided the target transputers are known. Each module is compiled for all required transputer types. The resulting library may be large and contain a certain amount of duplication.

Semi-optimised library build targeted at all transputer types

This is the simplest way to build a library that covers the full range of transputers. The user should compile each module to be included in the library for the following three general cases:

T2, TA and T8

The resulting library will be small in terms of the number of modules it will contain. Due to their generic nature the modules themselves may be bulky and because they contain only the base set of instructions, the execution time for the program will tend to be slower than a more optimised approach.

Optimised library

The transputer type determines the instruction set used for the compilation. Transputer classes TA and TB provide the basic instruction sets common to several transputer types. Transputer classes such as the T5 provide extended instruction sets but are targeted at fewer transputers than classes TA and TB.

The C compiler will attempt to use the **dup** instruction to produce better code. If this instruction would make the compilation more efficient but it is not present in the processor class being used for the compilation, a warning message is generated.

In order to build a library which is both generalised enough to work for all 32-bit transputers and is then optimised for modules which require extended instructions sets the following approach is recommended:

1 Compile all modules for class TA and type T8. This will provide modules which can be run on all 32-bit transputers.

2 Any modules which could be compiled more efficiently using the **dup** instruction, should also be compiled for type T5.

For 16-bit transputers, all modules should be compiled for T2. Any modules which use the **dup** instruction should also be compiled for type T3.

19.6 Error messages

This section lists each error and warning message which may be generated by the librarian. Messages are in the standard toolset format which is explained in section A.6.1.

19.6.1 Warning messages

filename **- bad format: symbol** *symbol* **multiply exported**

An identical symbol has occurred in the same file. There are three possibilities:

- The same file has been specified twice.
- The file was a library where previous warnings have been ignored.
- A module in the file has been incorrectly generated.

filename1 **- symbol** *symbol* **also exported by** *filename2*

An identical symbol has occurred in more than one module. If the linker requires this symbol, it will never load the second module.

19.6.2 Serious errors

bad format: *reason*

A module has been supplied to the librarian which does not conform to a recognised INMOS file format or has been corrupted.

filename - line number **- bad format: excessively long line in indirect file**

A line is too long. The length is implementation dependent, but on all currently supported hosts, is long enough to only be exceeded in error.

filename - line number **- bad format: file name missing after** *directive*

A directive (such as `INCLUDE`) has no file name as an argument.

filename - line number **- bad format: non ASCII character in indirect file**

The indirect file contains some non printable text. A common mistake is to specify a library or module with the `F` command line argument or the `INCLUDE` directive.

bad format: not a TCOFF file

The supplied file is not a library or module of any known type.

filename - line number **- bad format: only single parameter for** *directive*

The directive has been given too many parameters.

command line error *token*

An unrecognised token was found on the command line.

filename **- could not open for reading**

The named file could not be found/opened for reading.

filename1 - line number **- could not open** *filename2* **for reading**

The file name specified in an `INCLUDE` directive could not opened.

filename **- could not open for writing**

The named file could not be opened for writing.

filename **- must not mix linked and linkable files**

The librarian is capable of creating libraries from compiled modules or linked units, but it is illegal to attempt to create a library from both.

no files supplied

Options have been given to the librarian but no modules or libraries.

filename **- nothing of importance in file**

The file name specified in a library indirect file or in an `INCLUDE` directive was empty or contained nothing but white space or comments.

filename - line number **- only one file name per line**

More than one file name has been placed on a single line within an indirect file.

filename - line number **- unrecognised directive** *directive*

An unrecognised directive has been found in an indirect file.

20 `ilink` – linker

This chapter describes the linker tool **`ilink`** which combines a number of compiled modules and libraries into a linked object file. The chapter begins with a short introduction to the linker, explains the command line syntax and goes on to describe linker indirect files and the main linker options. The chapter ends with a list of linker messages.

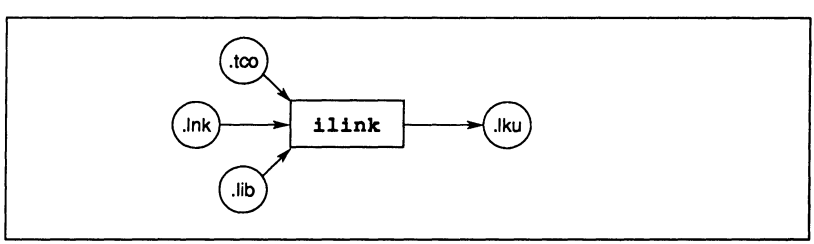
20.1 Introduction

The linker links a number of compiled modules and library files into a single linked object file, resolving all external references. The linker may be used to link object files produced by TCOFF compatible compilers including **`icc`**, by the librarian **`ilibr`**, or by the file format convertor **`icvlink`**. It can also be used with other TCOFF compatible compilers such as the occam 2 compiler **`oc`**. Code produced by the linker can be used as input to the configurer and collector tools to produce a bootable code file.

The linker can be driven directly from the command line or indirectly from a *linker indirect file*. This is a text file which contains a list of files to be linked, together with directives to the linker.

The linker is designed to accept input files in the *T*ransputer *C*ommon *O*bject *F*ile *F*ormat (TCOFF) supported by this release of the toolset. However, the linker can be directed to produce special format output files for use by the **`iboot`** or **`iconf`** tools used in previous releases of the toolset. Files for input to these tools are generated in *L*inker *F*ile *F*ormat (LFF).

The operation of the linker in terms of standard input and output file extensions is shown below.



Note: The linker does not support the pre-linking of files.

20.2 Running the linker

To invoke the linker use the following command line:

▶ **ilink** [*filenames*] {*options*}

where: *filenames* is a list of compiled files, library files, or files converted from previous toolsets using **icvlink**.

options is a list of any of the options given in tables 20.1 and 20.2.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

If an error occurs during the linking operation no output files are produced.

Examples of use:

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icollect hello.lku -t
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icollect hello.lku /t
iserver /sb hello.btl /se
```

In this example a compiled file is linked for the default T414 transputer, using the startup file **startup.lnk**. The example also shows the steps for compiling, booting and loading the program.

Option	Description
TA	Specifies target transputer class TA (T400, T414, T425, T800, T801, T805)
TB	Specifies target transputer class TB (T400, T414, T425)
T212	Specifies a T212 target processor.
T222	Specifies a T222 target processor. Same as T212.
M212	Specifies a M212 target processor. Same as T212.
T2	Same as T212, T222 and M212.
T225	Specifies a T225 target processor.
T3	Same as T225.
T400	Specifies a T400 target processor. Same as T425.
T414	Specifies a T414 target processor. This is the default processor type and may be omitted when linking for a T414 processor.
T4	Same as T414 (default).
T425	Specifies a T425 target processor.
T5	Same as T400 and T425.
T800	Specifies a T800 target processor.
T8	Same as T800.
T801	Specifies a T801 target processor. Same as T805.
T805	Specifies a T805 target processor.
T9	Same as T801 and T805.
H	Generates the linked unit in HALT mode. This is the default mode for the linker and may be omitted for HALT mode programs. This option is mutually exclusive with the 'S' option.
S	Generates the linked unit in STOP mode. This option is mutually exclusive with the 'H' option.
X	Generates the linked unit in UNIVERSAL mode. See section 20.4.2 below.

Table 20.1 `ilink` command line options

Option	Description
T	Specifies that the output is to be generated in TCOFF format. This format is the default format.
LB	Specifies that the output is to be generated in LFF format, for use with the **iboot** bootstrap tool and **iconf** configurer tool used in previous toolsets.
LC	Specifies that the output is to be generated in LFF format, for use with the **iconf** tool (supported by previous issues of the toolset).
F *filename*	Specifies a linker indirect file.
I	Displays progress information as the linking proceeds.
KB *memorysize*	Specifies virtual memory required in Kilobytes.
L	Loads the tool onto the transputer board and terminates.
ME *entryname*	Specifies the name of the main entry point of the program and is equivalent to the **#mainentry** directive (see below).
MO *filename*	Generates a module information file with the specified name.
O *filename*	Specifies an output file.
U	Allows unresolved references.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.
Y	Disables interactive debugging for occam code. Used only when linking in occam modules compiled with interactive debugging disabled.

Table 20.2 **ilink** command line options

20.2.1 Default command line parameters

A set of default command line options can be defined for the tool using the **ILINKARG** environment variable. Options must be specified using the standard command line format.

20.3 Linker indirect files

Linker indirect files are text files containing lists of input files and commands to the linker. Indirect files are specified on the command line using the '**F**' option.

Linker indirect files can contain filenames, linker directives, and comments. Filenames and directives must be on separate lines. Comment lines are introduced by the double dash ('--') character sequence and extend to the end of line. Comments must occupy a single line.

Indirect files can include other indirect files.

20.3.1 Linker directives

The linker supports six directives which can be used to fine tune the linking operation. Linker directives must be incorporated in indirect files (they cannot be specified on the linker command line) and are introduced by the hash ('#') character.

The six linker directives are summarised below and described in detail in the following sections.

Directive	Description
`#alias`	Defines a set of aliases for a symbol name.
`#define`	Assigns an integer value to a symbol name.
`#include`	Specifies a linker indirect file.
`#mainentry`	Defines the program main entry point.
`#reference`	Creates a reference to a given name.
`#section`	Defines the linking priority of a module.
Note: All symbol names are case sensitive.	

#alias *basename {aliases}*

The **#alias** directive defines a list of aliases for a given base name. Any reference to the alias is converted to the base name before the name is resolved or defined. For example, if a module contains a call to routine 'proc_a', which does not exist, then another routine 'proc_d' may be given the alias 'proc_a' in order to force the call to be made to routine 'proc_d'.

```
#alias proc_d proc_a
```

In the above example the reference to 'proc_a' is considered to be resolved. Modules may be loaded from the library for 'proc_d' but the linker will not attempt to search for library modules for 'proc_a'. If a procedure called 'proc_a' is found in any module then an error will result as the symbol will be multiply defined.

#define *symbolname value*

The **#define** directive defines a symbol and gives it a value. This value must either be an optionally signed decimal integer, or an unsigned hexadecimal integer. (If it is the latter it must be preceded by a # sign).

#include *filename*

The **#include** directive allows a further linker indirect file to be specified. Linker indirect files can be nested to any level. The following is an example of nested indirect files:

```
-- user's .lnk file:

userproc1.tco        -- module
#mainentry proc_a    -- main entry point directive
#include sub.lnk     -- nested indirect file

-- user's sub.lnk file:

userproc2.tco        -- further modules
userproc3.tco
hostio.lib           -- library
```

#mainentry *symbolname*

The **#mainentry** directive defines the main entry point of the program ie. the lowest level functions of the program. This directive is equivalent to the '**ME**' command line option. Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default.

The supplied indirect files for linking C programs define the C system main entry point.

`#reference` *symbolname*

The `#reference` directive creates a forward reference to a given symbol. This allows names to be made known to the linker in advance, or forces linking of library modules that would otherwise be ignored. The purpose is to allow the inclusion of library initialisation routines which might not otherwise be included. For example:

```
#reference so.open
```

The above example causes so.open to be included in the link, whether it is needed or not.

`#section` *name*

The `#section` enables the user to define the order in which particular modules occur in the executable code.

In order to use this directive the program modules must have been compiled using the compiler directive `#pragma IMS_linkage`. Details of the syntax can be found in section 11.3.11.

The linkage directive associates a section name with the code of a compilation module. A section name may take the default value "`pri%text%base`" or a name specified by the user.

The linker will place modules, associated with the section name "`pri%text%base`", first in the code of the linked unit, in the order in which these modules are encountered. When the linker directive `#section` is used this default condition is overridden. The modules identified by user defined section names will be placed first in the linked module, in the order in which the `#section` directives are encountered. These will be followed by any other modules in an undefined order at the end of the linked unit. For example:

```
#section first%section%name
#section second%section%name
```

In the above example any modules identified by `first%section%name` will be linked first, followed by modules identified by `second%section%name`, followed by all other modules not identified by a section name.

20.3.2 Linker startup files

The linker command line must include a linker startup file which specifies to the linker where to locate the runtime library. Two files are supplied: **startup.lnk** which should be used for all single transputer programs and multitransputer programs which use the full library; and **startrd.lnk** which should be used for *multi*transputer programs which use the reduced library.

Both link startup files are in standard linker indirect file format.

For mixed language programs that incorporate occam code a further linker indirect file may be included on the command line. One of three files may be selected, each file supports different target processor types, as shown below.

Linker indirect file	Target processors
occam2.lnk	T212/222/225/M212
occama.lnk	T400/414/425/TA/TB
occam8.lnk	T800/801/805

Each file contains a list of occam library files which may be required to be linked, but which are additional to those obviously referenced by the program. Depending on the other inputs and options specified on the command line the linker will select which libraries it requires from the supplied indirect file.

20.4 Linker options

The following sections describe the main command options which may be specified to the linker.

20.4.1 Processor types

A number of options are provided to enable the user to specify the target processor for the linked object file, see Table 20.1. Only one target processor or transputer class may be specified and this must be compatible with the processor types or transputer class used to compile the modules. (See section 5.3 for details of transputer classes).

If no target processor is specified, the processor type for the linked object file will default to a T414 processor type.

If any input file in the list is incompatible with the processor type in use, the link fails and an error is reported.

20.4.2 Error modes – options H, S and X

Three error modes are provided by the toolset for linking C programs with modules compiled using other INMOS language toolsets. All programs written exclusively in C are compiled by `icc` in UNIVERSAL mode.

The error modes provided for mixed language programming are as follows:

HALT	An error halts the transputer immediately.
STOP	An error stops the process and causes graceful degradation.
UNIVERSAL	Modules compiled in this mode may be run in either HALT or STOP mode depending on which mode is selected at link time.

Modules that are to be linked together must be compiled for compatible error modes. Table 20.3 indicates the compilation error modes which are compatible and the possible error modes they may be linked in.

Compatible error modes	`ilink` options
HALT, UNIVERSAL	`H`
STOP, UNIVERSAL	`S`
UNIVERSAL	`X`

Table 20.3 `ilink` error modes

Note: modules which have been compiled in UNIVERSAL error mode may be linked in this mode but the resulting linked unit will be treated by the `icconf` and `icollect` tools as if it had been linked in HALT mode.

The linker will produce an error if an input file is in a mode incompatible with the command line options or defaults.

20.4.3 TCOFF and LFF output files – options T, LB, LC

These three options enable the format of the linked unit output file to be changed. The linker will default to option `T` if none is specified.

Option '`T`' specifies that the linked unit is to be output in TCOFF format. This file may then be processed by other tools in the current toolset, for example, the configurer `icconf` and the collector `icollect`.

The 'LB' and 'LC' options specify that the linked unit is to be output in LFF format so that it is compatible with previous toolsets. The 'LB' option produces a file compatible with the iboot and iconf tools used by previous toolsets. The specified main entry point of the linked program is then available for bootstrapping by iboot or configuring by iconf.

The 'LC' option is used in mixed language programming for occam programs only. No main entry point need be specified.

When the 'LB' and 'LC' options are used the output file will not be compatible with the current toolset.

20.4.4 Display information – option I

This option enables the display of linkage information as the link operation proceeds.

20.4.5 Virtual memory – option KB

The KB option allows the user to specify how much memory the linker will use for storing the image of the users program. By default the linker will attempt to store the entire image in memory. In situations where memory is limited, an amount (>= 1Kbytes) may be specified. If the program is larger than the amount specified then the linker will use the host filing system as an intermediate store. A reduction in speed may be expected.

20.4.6 Main entry point – option ME

The ME option defines the main entry point of the program ie. the point from which linking will start. This option is equivalent to the #mainentry directive and takes as its argument a symbol name which is case sensitive.

Only one main entry point may be specified. If it is omitted the linker will select the first valid entry point in its input as a default.

20.4.7 Link map filename – option MO

This option causes a link map file to be produced with the specified name. A file extension should be specified as there is no default available. If the option is not specified a separate link map file is not produced.

20.4.8 Linked unit output file – option O

The name of the linked unit output file can be specified using the 'O' option. If the option is not specified the output file is named after the first input file given on the command line and a `.lku` extension is added. If the first file on the command line is an indirect file the output file takes the name of the first file listed in the indirect file.

Note: That because there is no restriction on the order in which files may be listed it is up to the user to ensure that his output file is named appropriately.

20.4.9 Permit unresolved references – option U

The linker normally attempts to resolve all external references in the list of input files and reports any that are unresolved as errors.

Sometimes it is desirable to allow unresolved external references, for example during program development. The 'U' option allows the link to proceed to completion by assuming unresolved references are to be resolved as zero. Warning messages may still be generated and the program will only execute correctly if such references are in fact redundant.

20.4.10 Disable interactive debugging – option Y

This option applies only to the occam modules in mixed language programs. The option directs the linker to select modules that use sequences of transputer instructions for i/o instead of a set of library routines.

20.5 Selective linking of library modules

Library modules that are compiled for incompatible processor types or error modes are ignored by the linker. This allows library modules to be selectively loaded for specific processor types or transputer classes.

The standard C run time library is supplied in several forms to cover the complete range of transputer types. User libraries that are likely to be used on different transputer types can adopt the same strategy.

Libraries are also selected for linking on the basis of previous usage. Modules that are used by several input files are linked in only once.

20.6 The link map file

A file containing a map of the code being linked will be generated if the command line option MO is specified.

The file is generated in text format and contains information which may assist the user during program debugging. The map contains information about two categories of input file; separate compilation units, and library modules. The following information is included:

- Details of the target processor.
- Details of the main entry point – its source and the amount of workspace and vector space used.
- A list of the linkage sections used indicating the number of words each occupies.
- A list of modules used, indicating the source of the module, error mode, address, size and the first reference used to call it. (Addresses are displayed as byte offsets from the start of the code).

Independent of whether the MO option is used, the module data and details of the target processor are always included in the linked unit output file in the form of a comment.

20.7 Using imakef for version control

The **imakef** tool may be used to simplify the linking of complex programs, particularly those which use libraries that are nested within other libraries or compilation units.

Note: for **imakef** to function the file extensions described in chapter 22 *must* be used.

20.8 Error messages

This section lists each error and warning message that can be generated by the linker. Messages are in the standard toolset format which is explained in appendix A.

20.8.1 Warning messages

filename **- bad format:** *reason*

The named file does not conform to a recognised INMOS file format or has been corrupted.

Size **bytes too large for 16 bit target**

The code part of the linked unit has exceeded the address space of the T212 derived processor family.

filename - symbol, **implementation of channel arrays has changed**

Only generated in mixed language programs where occam code is used that was compiled in LFF format using previous INMOS toolsets. LFF files are often generated so that the LFF configurer may be used, but it should be noted that channel arrays should not be used as parameters to configured procedures since they are implemented differently in the new occam compiler and the old configurer.

filename **- symbol** *symbol* **not found**

The specified symbol was not found in any of the supplied modules or libraries.

file1 **- usage of** *symbol* **out of step with** *file2*

May be generated when linking mixed language programs incorporating modules written using language such as occam which have a **#USE** *file* directive which causes the compiler to scan the file for details concerning certain program resources. It is therefore essential that this file be unchanged at link time. This diagnostic indicates that this is not the case. There are several possible causes:

file2 has been recompiled after file1, in which case *file1* requires recompiling.

The file that occurred in the **#USE** directive has been replaced by a different version of the file at link time.

The file that occurred in the **#USE** directive has not been supplied to the linker, but the linker has located a different version of a required entrypoint elsewhere.

The occam compiler may need to scan certain libraries, of which the user is unaware. Specifying one of the linker indirect files **occam2.lnk**, **occama.lnk** or **occam8.lnk** should take care of these 'hidden' libraries.

20.8.2 Errors

filename **- name clash with** *symbol* **from** *filename*

May be generated when linking mixed language programs.

In languages such as occam entrypoints may be scoped, i.e. extra information is associated with each symbol to indicate which version of that entry point it is. This allows programs to be safely linked even though there are several different versions of the same entrypoint occurring at different lexical levels within the program.

This error indicates that a language without occam-type scoping has been mixed with a scoped language and a name conflict has occurred between a scoped and non scoped symbol.

filename **- symbol** *symbol* **multiply defined**

The symbol, introduced in the specified file, has been introduced previously, causing a conflict. The same module may have been supplied to the linker more than once or there may be two or more modules with the same entry point or data item defined.

filename **- symbol** *symbol* **not found**

The specified symbol was not found in any of the supplied modules or libraries.

filename **- usage of** *symbol* **out of step with** *namefile*

May be generated when linking mixed language programs incorporating modules written using language such as occam which have a **#USE** *file* directive which causes the compiler to scan the file for details concerning certain program resources. It is therefore essential that this file be unchanged at link time. This diagnostic indicates that this is not the case. There are several possible causes:

file2 has been recompiled after file1, in which case *file1* requires recompiling.

The file that occurred in the **#USE** directive has been replaced by a different version of the file at link time.

The file that occurred in the **#USE** directive has not been supplied to the linker, but the linker has located a different version of a required entrypoint elsewhere.

The occam compiler may need to scan certain libraries, of which the user is unaware. Specifying one of the linker indirect files **occam2.lnk**, **occama.lnk** or **occam8.lnk** should take care of these 'hidden' libraries.

Serious errors

filename **- bad format:** *reason*

The named file does not conform to a recognised INMOS file format or has been corrupted.

filename - line number **- bad format: excessively long line in indirect file**

A line is too long. The length is implementation dependent, but on all currently supported hosts it is long enough so as only to be exceeded in error.

filename - line number **- bad format: file name missing after** *directive*

A directive (such as include) has no file name as an argument.

filename - line number **- bad format:** *directive* **invalid number**

A numeric parameter supplied to a directive does not correspond to the appropriate format.

filename **- bad format: multiple main entry points encountered**

A symbol may be defined to be the main entry point of a program by a compiler. Only one such symbol must exist within a single link.

filename - linenumber **- bad format: non ASCII character in indirect file**

The indirect file contains some non printable text. A common mistake is to specify a library or module with the '**F**' command line argument or the include directive.

filename **- bad format: not linkable file or library**

The linker expects that all files names presented without a preceeding switch (on the command line) or directive (in an indirect file) are either libraries or modules.

filename - line number **- bad format: only single parameter for** *directive*

The directive has been given too many parameters.

Cannot create output without main entry point

No main entry point has been specified.

Command line: 1k minimum for paged memory option

When using the KB option, the amount of memory used to hold the image of the program being linked is specified. There is a minimum size of 1k.

Command line: *token*

An illegal token has been encountered on the command line.

Command line: bad format number

A numerical parameter of the wrong format has been found.

Command line: image limit multiply specified

The command line option'KB' has been specified more than once.

Command line: 'load and terminate' option set, some arguments invalid

Options to load and terminate the linker have been specified in conjunction with other command line options. The linker cannot excute these options if it has been instructed to terminate first.

Command line: multiple debug modes

The command line option 'Y' has been specified more than once.

Command line: multiple error modes

More than one error mode has been specified to the linker.

Command line: multiple module files specified

The command line option 'MO' has been specified more than once.

Command line: multiple output files specified

The command line option 'O' has been specified more than once.

Command line: multiple target type

More than one target processor type has been specified to the linker.

Command line: only one output format allowed

The options 'T', 'LB' and 'LC' are mutually exclusive.

filename **- could not open for input**

The named file could not be found/opened for reading.

filename **- could not open for output**

The named file could not be opened for writing.

filename - line number **- could not open for reading**

The file name specified in an include directive could not opened.

could not open temporary file

The 'KB' option has been used in a directory where there is no write access or not enough disc space.

filename **- mode:** *mode* **- linker mode:** *mode*

The linker has been given a module to link which has been compiled with attributes incompatible with the options (or lack thereof) on the linker command line.

Multiple main entry points specified

The main entry point has been specified on the command line or in an indirect file more than once.

filename - line number - *directive* **not enough arguments**

The wrong number of arguments to a directive.

filename - **nothing of importance in file**

The file name specified in an include directive was empty or contained nothing but white space or comments.

Nothing to link

Various options have been given to the linker but no modules or libraries.

filename - line number - **only one file name per line**

More than one file name has been placed on a single line within an indirect file.

filename - line number - *directive* **too many arguments**

The wrong number of arguments to a directive.

Unknown error modes not supported in the LFF format
Unknown processors not supported in the LFF format

When generating LFF format files, certain constructs will have no representation. For example processor types that have come into existence since the LFF format was defined.

filename - line number - **unrecognised directive** *directive*

An unrecognised directive has been found in an indirect file.

20.8.3 Embedded messages

Tools that create modules to be linked with **ilink** may embed "messages" within them. Three levels of severity exist; serious, warning, and message. The documentation of the appropriate tool should be consulted for more information. The format of these messages is as follows:

Serious - ilink - *filename* - **message:** *message*

Warning - ilink - *filename* - **message:** *message*

Message - ilink - *filename* - *message*

21 `ilist` – binary lister

This chapter describes the binary lister tool **ilist**, which takes an object file and displays information about the object code in a readable form. The chapter provides examples of display options and ends with a list of error messages which may be generated by **ilist**.

21.1 Introduction

The binary lister tool **ilist** reads an object code file, decodes it, and displays useful information about the object code on the screen. The output may be redirected to a file. Command line options control the category of data displayed.

The **ilist** tool can decode and display object files produced by the Parallel C compiler **icc**, the linker, librarian, file convertor, configurer and collector tools, and by other TCOFF compatible compilers such as the occam 2 compiler **oc**. Text files, file formats produced by **ieprom** and dynamically loadable modules can also be displayed using **ilist**. Files already in editable ASCII format are listed without further processing.

The **ilist** tool will also list compilation and linked units in *L*inker *F*ile *F*ormat (LFF) which was used by previous INMOS toolsets.

Object code files reflect the modular structure of the original source. Single unit compilations produce a file containing a single object module, whereas units containing many compilations, such as libraries and concatenations of modules, produce object files with as many object modules. The data produced by **ilist** reflects the modular composition of object files.

21.2 Data displays

There are several categories of data that can be displayed. Categories are selected by options on the command line.

The main categories are:

- *Symbol data* – symbol names in each module. Information is displayed in tabular form.
- *External reference data* – names of external symbols used by each module. Information is displayed in tabular form.
- *Module data* – data for each module including target processor, compilation mode, and module file name.
- *Code listing* – code contained in each module, displayed in hexadecimal format.
- *Procedural data* – for external occam calls only.
- *Index data* – the content of library indexes.

Except where indicated, the examples used in this chapter show the output generated for a compiled `.tco` file generated by `icc`.

21.3 Running the lister

To invoke the binary lister use the following command line:

▶ **`ilist`** {*filenames*} {*options*}

where: *filenames* is a list of one or more files to be displayed.

options is a list of one or more of the options given in Table 21.1. Options will only be applied to files of the appropriate file type.

> Options must be preceded by '-' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Example:

```
ilist hello.tco - a   UNIX based toolsets
ilist hello.tco / a   MS-DOS and VMS based toolsets
```

In this example `ilist` is being instructed to display all the symbol data for the file `hello.tco`.

Option	Description
A	Displays all the available information on the symbols used within the specified modules.
C	Displays the code in the specified file as hexadecimal. This option also invokes the 'T' option by default.
E	Displays all exported names in the specified modules.
H	Displays the specified file(s) in hexadecimal format.
I	Displays full progress information as the lister runs.
L	Loads the tool onto the transputer board and terminates.
M	Displays module data.
N	Displays information from the library index.
O *filename*	Specifies an output file. If more than one file is specified the last one specified is used.
P	Displays any procedural interfaces found in the specified modules.
R *reference*	Displays the library module(s) containing the specified reference. This option is used in conjunction with other options to display data for a specific symbol. If more than one library file is specified the last one specified is used.
T	Displays a full listing of a file in any file format.
W	Causes the lister to identify a file. The filename (including the search path if applicable) is displayed followed by the file type. This is the default option.
X	Displays all external references made by the specified modules.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

Table 21.1 `ilist` command line options

ilist will attempt to identify the file type by its contents. If filenames only are supplied, **ilist** uses the default option '**W**'.

Table 21.2 lists the available options and indicates which file formats they may be used to list. The table also lists the file types it is recommended to use with each option, in order of usefulness.

Option	Permitted file format	Recommended usage
H	Any format	
O	Any format	
T	Any format	
W	Any format	
A	TCOFF only	**.lib, .tco, .lku**
C	TCOFF only	**.tco, .lku, .lib**
E	TCOFF only	**.lib, .tco, .lku**
M	TCOFF only	**.tco, .lku, .lib**
N	TCOFF libraries only	**.lib**
P	TCOFF only	**.lib, .tco, .lku**
R	TCOFF libraries only	**.lib**
X	TCOFF only	**.lib, .tco, .lku**

Table 21.2 Recommended options

ilist sends its output to the host standard output stream, normally the terminal screen. Facilities available on the host system may allow you to redirect the output to a file, or send it to another process, such as a sort program. For details of these facilities consult the documentation for your system.

21.3.1 Default command line parameters

A set of default command line options can be defined for the tool using the **ILISTARG** environment variable. Options must be specified using the standard command line format.

21.4 Specifying an output file – option O

The O option enables the user to redirect the display data to an output file. If more than one output file is specified on the command line then the last one specified is used. File extensions should be specified; defaults are not assumed.

Display options are described in the following sections.

21.5 Symbol data – option A

This option displays all the available information about the symbols used within the specified modules. A tabular format is used. The data produced by this display may require skilled interpretation.

The following information is given:

- Symbol name.
- Section attributes, if applicable.
- Symbol attributes.
- The number of the symbol within the module.
- Module name.
- Target processor.
- Error mode.

Certain attributes apply only to symbols which are section names. If they are applicable, these attributes are indicated by the following nomenclature and displayed as a character string:

R – Read section.
W – Write section.
X – Execute section.
D – Debug section.
V – Virtual section.

Attributes for all symbols, including section names, are also indicated by a character string, using the following nomenclature:

L – Symbol local to the module.

E – Symbol exported from the module.

I – Symbol imported to the module.

W – Weak attribute, indicates that the symbol takes the value 0 when not defined.

C – Conditional attribute, indicates that the first value given to the symbol is always used.

U – Unindexed, indicates that the symbol is not present in the library index.

P – Provisional attribute, indicates that the last value given to the symbol is always used.

O – Indicates that the symbol is an origin symbol. The origin symbol is used by the linker to check the origin of the module.

Symbol attributes are displayed immediately after the section attributes, and each attribute is displayed at a specific position in the string. Attributes which are not present are indicated by a hyphen '-'.

The position of each attribute in the string is as follows:

```
RWXDV LEIWCUPO
```

Figure 21.1 provides an example of the symbol data displayed for a single `.tco` file by the '**A**' option.

```
module%table%base       ----V -E------ 0    hello.c          T414 X
module%number           ----- L------- 1    hello.c          T414 X
static%base             ----V -E------ 2    hello.c          T414 X
local%static            ----- L------- 3    hello.c          T414 X
%lsb                    ----- L------- 4    hello.c          T414 X
text%base               R-X-- -E------ 5    hello.c          T414 X
local%text              ----- L------- 6    hello.c          T414 X
main                    ----- -E------ 7    hello.c          T414 X
_IMS_printf             ----- --I----- 8    hello.c          T414 X
first%init%block        ----- -E--CU-- 9    hello.c          T414 X
next%init%block         ----- -E---UP- 10   hello.c          T414 X
next%init%block         ----- -E---UP- 11   hello.c          T414 X
```

Figure 21.1 Example output produced by the **A** option.

21.6 Code listing – option C

This option produces a full listing of the code in the same format as that generated by the '**T**' option, but with the addition of a hex listing of the code at each **LOAD_TEXT** directive. This option *may* be accompanied by the '**T**' option; if the '**T**' option is not specified it is supplied automatically.

The hex listing consists of the address followed an ASCII hex dump of the code, followed by an a representation of the code in ASCII characters. The format is as follows:

address ASCII hex ASCII characters

where: address is the address of the first byte on the line, expressed as an offset from the start of the module.

ASCII hex is the hex representation of the code

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable as an ASCII character it is replaced by a dot (.).

Figure 21.2 shows an example of the output produced by listing a `.tco` file. The example shows only the hex listing for an individual **LOAD_TEXT** entry; normally this appears embedded within the full display produced by the 'T' option.

```
000000DB 4521FB71 219222F0 68656C6C 6F20776F      E!.q!.".hello wo
000000EB 726C640A 00202020                        rld..
```

Figure 21.2 Example display produced by the **C** option.

21.7 Exported names – option E

The output from this option is in a tabular format. It consists of a list of names exported by the modules. This option also displays any globally visible data.

The following information is given by the display:

- Exported name.
- The name of the module in which the exported name is found.
- Language used.
- Target processor.
- Error mode.

Figure 21.3 shows the output produced by listing a `.tco` file.

```
main                    -> hello.c          ANSI_C      T414 X
```

Figure 21.3 Example display produced by the **E** option.

21.8 Hexadecimal/ASCII dump – option H

This option provides a display of the specified files in hexadecimal and ASCII format. The option does not attempt to identify file types and may be used to display any files which the lister has previously identified incorrectly.

The output takes the form of a hexadecimal representation of the whole of the file content. The display has a similar appearance to that produced by the **C** option, however, the **C** option only functions on code found within the file.

Each module is displayed as a contiguous block of lines, where each line has the format:

address *ASCII hex* *ASCII characters*

where: *address* is the address of the first byte on the line, expressed as an offset from the start of the module.

ASCII hex is the hex representation of the characters found.

ASCII characters are the ASCII characters corresponding to the hex code.

In all cases code is read from left to right. If a value is not printable it is replaced by a dot (.).

Figure 21.4 shows the display produced by listing the text file **hello.c** using the 'H' option.

```
00000000 23696E63 6C756465 203C7374 64696F2E      #include <stdio.
00000010 683E0A0A 696E7420 6D61696E 28290A7B      h>..int main().{
00000020 0A202070 72696E74 66282268 656C6C6F      . printf("hello
00000030 20776F72 6C645C6E 22293B0A 7D0A           world\n");.}.
```

Figure 21.4 Example display produced by the **H** option.

21.9 Module data – option M

This option displays any header information which is present. This may include version control data, general comments that may have been appended to the file during use of the toolset and copyright information. The data is displayed for individual modules in the object file and includes:

- Module name.
- Transputer type and compilation error mode.
- Language type.
- Version control data.
- Comments inserted by the toolset, for example, copyright clauses.

Data is displayed in separate blocks for each module. Some of the data is also used by other tools in the toolset, for example, some comments are used by the debugger tool **idebug** while version information is used by some tools for compatibility testing.

When linked units are displayed using this option, a long comment will be displayed. This comment gives details of the allocation of memory to each separately compiled code and library module used in the linked module. The following information is given in tabular format:

- Code type - Separately compiled code (SC) or library module (LIB).
- Module name.
- Address offset in linked module.
- Start address.
- End address.
- Reference in library (if applicable) used to locate the relevant library module.

The example in figure 21.5 shows the listing displayed for a `.tco` file.

```
MODULE:  ANSI_C          T414 X
VERSION: icc hello.c
```

Figure 21.5 Example display produced by the **M** option.

21.10 Library index data – option N

This option is used to list library indexes. The data is given in a tabular format. For each entry in the index the following information is given:

- The address of the module in the library.
- The symbol name.
- The language the module is written in.
- The target processor type.
- The error mode used.

Figure 21.6 shows the output produced by the '**N**' option on a library file consisting only of the compiled object module for a simple 'Hello world' program.

```
000000B2 main                    ANSI_C        T414 X
```

Figure 21.6 Example display produced by the **N** option.

21.11 Procedural interface data – option P

This option displays procedural interface information for external occam functions and procedures only. The following information is displayed:

- Target processor.
- Error mode.
- Language used.
- Amount of workspace used by the procedure.
- Amount of vector space used by the procedure.
- Parameters used by the procedure.
- Data type of parameters.
- Channel usage, if applicable.

A channel marked with an **?** is an *input* channel to the code of that entry point, and a channel marked with **!** is an *output* channel.

When a library file is listed this will be indicated by the words '**INDEX ENTRY mode:**' rather than '**DESCRIPTOR mode**'.

Figure 21.7 shows the procedural data output for a simple external occam procedure.

21.12 Specify reference – option R

This option is used in conjunction with any of the other options to select a specific symbol within a named library. All library modules that export the symbol are displayed. The exact format of the display depends on the main display option with which '**R**' is used.

```
DESCRIPTOR mode: T414 H    language:  lang: OCCAM <ORIGIN DESCRIPTOR>
DESCRIPTOR mode: T414 H    language:  lang: OCCAM
ws: 76 vs: 128
PROC simple(CHAN OF SP fs,
CHAN OF SP ts,
[]INT memory)
  SEQ
    fs?
    ts!
:
```

Figure 21.7 Example display produced by the **P** option.

Note: Symbol names must be specified in the correct case.

21.13 Full listing – option T

This option displays all *data* found in the input file. Provided that **ilist** recognises the file type, the file is decoded in its own format. Text file are displayed as text and unrecognised file types are displayed as a hexadecimal dump.

Data is not displayed in a tabular form but is output in the sequence in which it is found in the module.

The display formats are tailored to each file format and are intended for diagnostic support and analysis. The display generates large amounts of data and may require skilled interpretation.

Note: The full data listing of a *configured* object file also shows how the processes are mapped onto a transputer system.

Figure 21.8 shows part of the full data output for a compiled object file.

```
00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP BIT32 MS=18 X lang: ANSI_C ""
00000010 VERSION tool: icc origin: hello.c
0000001D SECTION VIR EXP "module%table%base" id: 0
00000033 SET_LOAD_POINT id: 0
00000036 SYMBOL LOC "module%number" id: 1
00000047 DEFINE_LABEL id: 1
0000004A ADJUST_POINT 1
0000004E SECTION VIR EXP "static%base" id: 2
0000005E SET_LOAD_POINT id: 2
00000061 SYMBOL LOC "local%static" id: 3
00000071 DEFINE_LABEL id: 3
00000074 SYMBOL LOC "%lsb" id: 4
0000007C DEFINE_SYMBOL id: 4 SS:0+SV:3
00000084 ADJUST_POINT 1
00000088 SECTION REA EXE EXP "text%base" id: 5
00000096 SET_LOAD_POINT id: 5
00000099 SYMBOL LOC "local%text" id: 6
000000A7 DEFINE_LABEL id: 6
000000AA LOAD_EXPR size: 4 SV:1
000000AF COMMENT COPY bytes: 5
000000B9 SYMBOL EXP "main" id: 7
000000C1 DEFINE_SYMBOL id: 7 SV:6+4
000000C9 SYMBOL IMP "_IMS_printf" id: 8
000000D8 LOAD_TEXT bytes: 24
000000F3 LOAD_PREFIX size: 6 AP(SV:8-LP) instr: j
000000FC LOAD_TEXT bytes: 2
00000101 SYMBOL EXP CON UNI "first%init%block" id: 9
00000115 DEFINE_LABEL id: 9
00000118 SYMBOL EXP UNI PRO "next%init%block" id: 10
0000012B DEFINE_LABEL id: 10
0000012E KILL_ID id: 10
00000131 SYMBOL EXP UNI PRO "next%init%block" id: 11
00000144 DEFINE_LABEL id: 11
00000147 LOAD_EXPR size: 4 SV:11-LP
0000014E LOAD_TEXT bytes: 3
00000154 LOAD_PREFIX size: 6 SV:4 instr: ldnlp
0000015A LOAD_TEXT bytes: 3
00000160 LOAD_PREFIX size: 3 SV:1 instr: stnl
00000166 LOAD_TEXT bytes: 5
0000016E COMMENT COPY bytes: 33
00000194 END_MODULE
```

Figure 21.8 Example display produced by option **T** for a `.tco` file.

21.14 File identification – option W

This option causes the lister to identify the file. The filename is displayed along with the file type and the full pathname is also given if applicable. This is the default command line operation if no options are given.

Table 21.3 indicates the different file types identified by the lister.

File format	Default extension	Listed file type
TCOFF compiled unit	.tco	TCOFF LINKABLE UNIT
TCOFF compiled library unit	.lib	TCOFF LINKABLE UNIT LIBRARY
TCOFF linked unit	.lku	TCOFF LINKED UNIT
TCOFF linked library unit	.lib	TCOFF LINKED UNIT LIBRARY
Configuration binary	.cfb	CONFIGURATION BINARY
Core dump	.dmp	CORE DUMP FILE
Network dump	.dmp	NETWORK DUMP
LFF file	.cxx, .txx	LFF SC
LFF library	.lib	LFF LIBRARY
Extracted SC	.rxx	EXTRACTED SC
iboot program	.bxx	BOOTABLE PROGRAM (**iboot**)
Extracted program	.btl	BOOTABLE PROGRAM
Empty file	–	EMPTY FILE
Text files	–	TEXT FILE
None of the above	–	UNKNOWN BINARY FORMAT

Table 21.3 File types recognised by **ilist**

SC files are separately compiled files.

LFF files are separately compiled or linked files in LFF format.

Extracted files are files which have been compiled and developed to be dynamically loaded onto a transputer system.

iboot programs are programs which have had a bootstrap added by the **iboot** tool, supported by previous issues of the toolset.

21.15 External reference data – option x

This option displays a list of all the code and data symbols imported by the modules specified to the lister, i.e. it lists their external references. External references are references to separately compiled units. For C programs the option will also display any external references to globally visible data.

The output from this option is in a tabular format. It consists of a list of external references and their associated modules.

The following information is displayed:

- External reference i.e. name of the separately compiled unit.
- The name of the module in which the external reference exists.
- Language used.
- Target processor.
- Error mode.

Figure 21.9 shows an example of the output generated for a compiled object file.

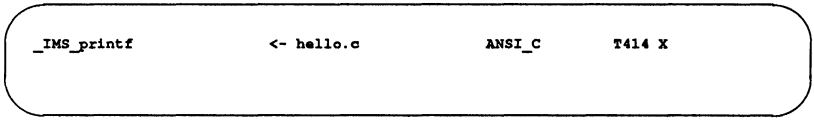


Figure 21.9 Example display produced by the **X** option.

21.16 Error messages

This section list each error ane warning message that can be generated by the lister. Messages are in the standard toolset format which is explained in appendix A.

21.16.1 Warning messages

filename - *reason*

The named file does not conform to a recognised INMOS file format or has been corrupted.

21.16.2 Serious errors

filename - **bad format:** *reason*

The named file does not conform to a recognised INMOS file format or has been corrupted.

filename **- could not open for input**

The named file could not be opened for reading.

filename **- could not open for output**

The named file could not be opened for writing.

filename **- file type does not correspond to command line options**

The options given to the lister apply to formats dissimilar to the format of the file being read.

must supply additional TCOFF options with reference *reference*

The required format of the listing has not been specified.

filename **- no entry for** *reference* **in library index**

The specified reference cannot be found in the library index.

parsing command line *token*

An unrecognised token was found on the command line.

filename **- unexpected end of file**

The named file does not conform to a known INMOS file format or has been corrupted.

22 `imakef` – Makefile generator

This chapter describes the Makefile generator **`imakef`** that creates Makefiles for input to MAKE programs. It explains how the tool can be used to create Makefiles and describes the special file naming conventions that allow **`imakef`** to create Makefiles for mixtures of code types. The chapter describes the format of Makefiles generated by **`imakef`** and ends with a list of error messages.

22.1 Introduction

MAKE programs automate program building by recompiling only those components that have been changed since the last compilation. To do this they read a **Makefile** which contains information about the interdependencies of files with one another, along with command lines for rebuilding the program.

`imakef` creates Makefiles for all types of toolset object files using its built in knowledge of how files referenced within the target file depend on one another. It is intended to be used with all INMOS language toolsets that generate TCOFF object code. Its mode of operation for different languages is controlled by command line options. The Makefile is generated in a standard format for input to most MAKE programs.

Makefiles created using **`imakef`** are compatible with many public domain and proprietary MAKE programs. The following MAKE programs are directly compatible:

- Borland **`MAKE`**.
- UNIX **`MAKE`**.
- GNU **`MAKE`**.

However, Microsoft **`MAKE`** is not compatible.

The source of **`imakef`** is supplied withg the toolset so that it can be modified for use with other MAKE programs.

22.2 How imakef works

imakef operates by working back from the target file to determine its dependences on other files, using its knowledge of inputs and outputs of each tool and the compilation architecture of the toolset. For example, compiled object files must be created from language source files using the compiler. In a similar way linked files must be generated from compiled files, and bootable files from linked units or configuration data files. **imakef** works back from the target file, determining file dependencies and creating commands to recompile the code where necessary.

imakef identifies files and file types by a special set of file extensions which specify the transputer type and error mode and allow it produce Makefiles for mixed module combinations. Note that these extensions differ from the standard toolset defaults. The conventions are described in section 22.4.

Note: In order for **imakef** to work correctly the special file naming conventions must be followed at all stages of program development for all types of target file.

22.3 Target files

The following table lists the types of C object files for which **imakef** can create Makefiles. The file extensions required by **imakef** are also given for each of the files (see section 22.4).

File type	Extension
Compiled code.	.t*xx*
Linked code.	.c*xx*
Bootable code for single transputer programs.	.b*xx*
Bootable code for multitransputer programs.	.btl
Dynamically loadable code.	.r*xx*
Libraries.	.lib
Configuration binary files.	.cfb

22.4 File extensions for use with imakef

A special set of file extensions must be used for source and object files during a program's development if **imakef** is to be used to automate any part of the build. The file extensions specify to **imakef** the transputer target and execution error mode for each of the program modules, and extend the scope of the default toolset file naming conventions.

The naming convention is based on a three-character file extension which identifies different types of source and object files. Some object files use the second and third characters to identify the transputer type and execution error mode. This form is used for compiled code, linked units, bootable files, and non-bootable files.

The main extensions are shown in figure 22.1 in relation to toolset program development.

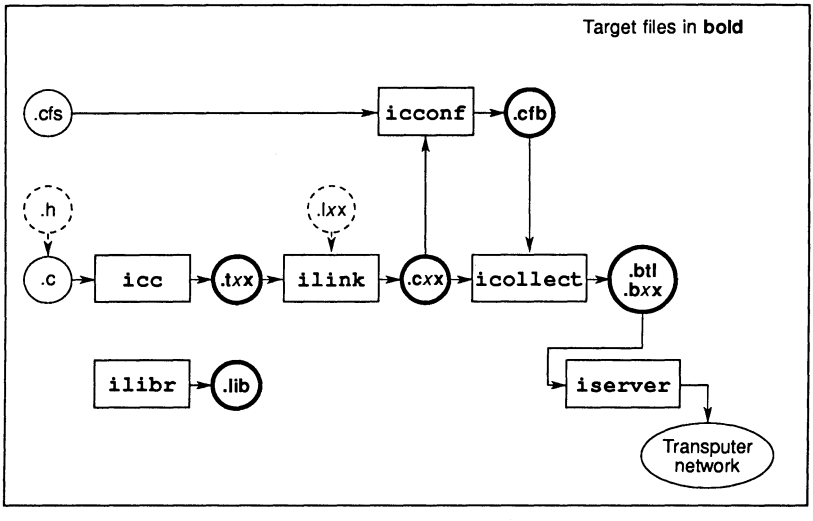


Figure 22.1 Main target files showing file extensions required

22.4.1 Transputer types and error modes

In the **imakef** system some object files use the second leter of the extension to designate the transputer target. The third letter represents the error mode which in C is always UNIVERSAL, designated by the letter '**x**'. For example, **.t4x** refers to a compiled C module targetted for the T4 transputer class.

Values that can be taken by the second character and their meanings are listed below.

Character	Transputer types supported
2	T212, T222, M212
3	T225
4	T414
5	T425, T400
8	T800
9	T805, T801
a	Class TA
b	Class TB

Error modes in mixed language programs

Object code generated by some other INMOS language toolsets can be compiled in two other error modes, namely HALT and STOP. These are represented by the letters '**h**' and '**s**' respectively. For example, **.t8h** refers to a foreign language module targetted for the T800 transputer in HALT error mode.

C object code generated in UNIVERSAL mode can be linked with HALT or STOP code generated by other languages by specifying the appropriate linker option.

For further information about the standards adopted for file extensions see section A.5.

22.5 Linker indirect files

Linker indirect files must be written for all linked units on which **imakef** is to be used. Linker indirect files define the components of the linked unit to **imakef** and provide a starting point for determining file dependencies.

Linker indirect files must be named after the linked unit to which they relate and must carry the **.lnk** extension.

22.6 Running the Makefile generator

The **imakef** program takes as input a list of files generated by tools in the toolset and generates a Makefile for each of the input files. Each output file is named after its target filename stem with a **.mak** extension (if no output file is specified on the command line).

Note: For correct operation with all C programs **imakef** *must* be invoked with the 'C' option and the constituent modules of each linked unit must be listed in an appropriate linker indirect file of the correct name.

To invoke **imakef** use the following command line:

▶ **imakef** {*filenames*} {*options*}

where: *filenames* is a list of target files for which Makefiles are to be generated. If more than one file is specified the single Makefile generated will generate all of the specified files.

options is a list, in any order, of one or more options from Table 22.1.

> Options must be preceded by '-' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

22.6.1 Example of use

```
imakef hello.b4x -c          (UNIX based toolsets)
imakef hello.b4x /c          (MS-DOS and VMS based toolsets)
```

This creates the Makefile **hello.mak** which when used as input to **MAKE** generates the bootable file **hello.b4x**.

22.6.2 Disabling debug data

Two options disable the creation of debug data.

Option	Description
C	Specifies that the list of files to be linked is to be read from a linker indirect file. This option *must* be specified for correct C operation.
D	Disables the generation of debugging information. The default is to compile with full debugging information.
I	Displays full progress information as the tool runs.
L	Loads the tool onto the transputer board and terminates.
O *filename*	Specifies an output file. If no file is specified the output file is named after the target file and given the **.mak** extension.
R	Writes a deletion rule into the Makefile.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.
Y	Disables interactive (breakpoint) debugging in the target compilation. The default is to compile with full breakpoint debugging information.

Table 22.1 **imakef** options

The 'D' option disables the generation of all debugging information in the target file. If this option is used the resulting target code cannot be debugged.

The 'Y' option disables only the data required for interactive (breakpoint controlled) debugging. If this option is given no breakpoint debugging operations can be used on the final program. Post-mortem debugging is unaffected.

22.6.3 Removing intermediate files

Intermediate files can be removed in final program build by specifying the 'R' option. This option adds a *delete* rule to the Makefile which directs MAKE to remove all intermediate files once the program is built. The delete operation is only honoured if MAKE is subsequently invoked with **DELETE** option.

22.7 imakef examples

This section contains two examples of the use of **imakef**. The first example shows how to create a Makefile for a multi-module program running on a single transputer and the second example shows how to create a Makefile for a configured program.

Both programs are supplied in the **imakef** examples subdirectory.

22.7.1 Single transputer program

The example program is made up of three files:

```
main.c
hellof.c
worldf.c
```

imakef needs to know the names of the main components of the program, and looks for the associated linker indirect file **hello.lnk**:

hello.lnk must contain the following text:

```
main.t4x
hellof.t4x
worldf.t4x
#include startup.lnk
```

Note the use of the **.t4x** extension rather than **.tco**. This is because **imakef** needs to work out the required processor type.

The standard C startup linker indirect file **startup.lnk** is also included. The inclusion of this file is standard for all C programs and directs **imakef** to include the C libraries.

To create the Makefile use the command:

```
imakef hello.b4x -c
```

Note the use of the **.b4x** extension instead of **.btl**. Using this form of extension informs **imakef** that we wish to create a bootable program for a single transputer without the aid of the configurer.

The Makefile **hello.mak** is created.

22.7.2 Multitransputer program

This example program uses the configurer to place linked units on two processors. The program is made up of the following files:

```
master.c
mult.c
multi.cfs
```

The **.cfs** file is the configuration description file. It places 2 linked units on 2 processors, using the following statements:

```
use "master.c8x" for master;
use "mult.c4x" for mult;
```

Note the use of the **.cxx** form of extension instead of the toolset default extension for linked units **.lku**.

imakef reads the **.cfs** file and determines that the program is made up of two linked units, each of which must have an associated linker indirect file, namely, **master.lnk**, and **mult.lnk**.

The two linker indirect files files must contain the following text:

master.lnk:

```
master.t8x
#include startup.lnk
```

mult.lnk:

```
mult.t4x
#include startrd.lnk
```

Again note the use of the **.txx** form of extension. A startup linker indirect file is again included in each file to access the libraries. Note the use of the startup file **startrd.lnk** in the second file which accesses the *reduced* library. This library can be used by **mult.t4x** because the module does not require host access.

To create the Makefile use the following command:

```
imakef multi.btl -c
```

The **.btl** extension informs **imakef** that the target is a configured program, to be built from a configuration description file called **multi.cfs**.

The Makefile **multi.mak** is created.

22.8 Format of Makefiles

Makefiles essentially consist of a number of *rules* for building all the parts of a program. Each rule contains two main elements: a definition of the file's dependencies in a format acceptable to MAKE programs; and the command to recreate the file on a specific host. All Makefiles also contain macros which define command strings and option combinations.

22.8.1 Macros

All Makefiles created by **imakef** include a set of macro definitions inserted at the head of the file.

Macros define strings which are used to call the compiler, the configurer, the linker, the librarian, the collector, and the eprom formatter tools, and fixed combinations of options for these tools.

Macros are provided so that customised versions of the toolset commands, and specific combinations of options, can be easily incorporated. Existing macros can be modified for specific host environments, and new macros created, by editing the Makefile.

The full set of macros defined by **imakef** can be found by consulting any Makefile created by the tool.

22.8.2 Rules

Rules define the dependencies of object files on other files and specify *action strings* to build those files. For example:

```
config.btl:  config.cfg prog.c4h
$(CONFIG) config -r $(CONFOPT) -o config.btl
```

This rule first defines the target as the bootable program **config.btl**, which is dependent on the configuration description file **config.cfg** and the linked file **prog.c4h**, and then specifies the command that must be invoked to build it.

The first rule in all Makefiles is for the main target. Succeeding rules define subcomponents of the main target, and are listed hierarchically.

Action strings

Action strings define the complete command line needed to recreate a specific file. The format is similar for all tools and consists of a call to the tool via a predefined macro, a fixed set of parameters, a list of command line options, probably also via a macro, and the output filename. (The output file is specified on the command line so that the rebuilt file is always written to the directory that contains the source.)

22.8.3 Delete rule

The delete rule directs MAKE to remove all intermediate object files once the program has been built. It consists of a single labelled action string which invokes the host system 'delete file' command. Deletion is only performed if MAKE is subsequently invoked with the DELETE option.

The delete rule is appended to the Makefile by specifying the **imakef** '**R**' option.

22.8.4 Editing the Makefile

Makefiles created by the **imakef** tool can be edited for specific requirements. For example, new macros can be added and new rules defined for compiling and linking code written in other languages.

Adding options

imakef generates action strings which have the minimum of options for each tool. In most cases additional options are unnecessary or may be specified using compiler directives. To modify the set of default options for a particular tool simply edit the appropriate macro in the Makefile.

For example, if debugging data is to be enabled for all invocations of the compiler the compiler 'G' option would be added to the **CCOPT** macro which defines the standard combination of options for invoking the compiler. Alternatively a new macro containing only the 'G' option could be defined and added to each compiler action string.

22.9 Error messages

imakef generates error messages of severities *Warning* and *Error*. Messages are displayed in standard toolset format.

Cannot have a makefile

The file specified on the command line is not one for which **imakef** can generate a Makefile. **imakef** can only create Makefiles for object files and bootable files.

Cannot open "*filename*" :*reason*

The file specified as the output file cannot be opened for writing by the program, for the reason given.

Cannot write linker command file

The linker command file cannot be opened for writing by the program.

Command line is invalid

An incorrect command line was supplied to the program. Check the syntax of the command and try again.

Error whilst reading

A file system error has occurred whilst reading the source.

#IMPORT references are illegal in configuration text

At the given line number in the file there is a reference to the **#IMPORT** directive, which is illegal for configuration source.

#INCLUDE may not reference a library

The **#INCLUDE** directive is being used to reference a file with the **.lib** extension.

#INCLUDE may not reference binary files

The **#INCLUDE** directive is being used to reference a file containing compiled code.

Incomplete compiler directive

At the given line number in the file there is an invalid compiler directive.

Library on PATH "*pathname*" also exists in the current directory

A library with the specified name has been found on the current search path and in the current directory.

Malloc failed

The program has failed while trying to dynamically allocate memory for its own use. Try using a transputer board with more memory. If the program is being run on the host it may be possible to increase the memory available using host commands.

Options are incorrectly deliminated

The terminating bracket which determines the options in a library build file, is missing at the given line number.

Source file does not exist

The referenced source file does not exist on the system.

Target is not a derivable file

The specified file cannot be generated by the toolset.

Tree checking failed - no output performed

The tree of files has been found to be invalid and unusable for generating Makefile. This message always follows a message indicating what is wrong with the tree. The most common reason for this error is the presence of cyclic references in the source.

"*filename*" **unknown/illegal file reference**

A compiler directive is attempting to reference the wrong type of file.

Writing file

An host system error occurred while the file was being written.

23 `iserver` – host file server

This chapter describes the host file server **`iserver`** which loads application programs onto transputer networks and provides runtime access to the host.

23.1 Introduction

The host file server **`iserver`** performs two functions:

- Loads bootable programs onto transputer hardware
- Provides the runtime environment which allows the program to talk to the host.

At the application program level, all communication with the host file server is through the standard i/o libraries. The host file server provides an intermediate interface through which the i/o functions can communicate with any of the supported hosts. The interface is based on a fixed protocol and is implemented by an underlying set of functions written in C. A description of the protocol and definitions of the functions can be found in appendix E.

23.1.1 Loadable programs

Before a program can be loaded onto a transputer network it must be compiled and linked. It may then be made bootable using the collector tool **`icollect`**. If no output file was specified when the program was built the loadable file will have a **`.btl`** file extension if the default extension is used. If **`imakef`** has been used to build the program the file will have an extension of the form **`.bxx`**. For further details of the file extension system used by **`imakef`** see sections 22.4 and A.5.

23.2 Running the server

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

▶ **`iserver`** *bootablefile* {*options*}

where: *options* is a list of one or more options from table 23.1.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
SA	Analyses the root transputer and peeks 8K of its memory.
SB *filename*	Loads the program contained in the specified file.
SC *filename*	Copies the specified file to the root transputer link.
SE	Terminates the server if the transputer error flag is set.
SI	Displays progress information as the program is loaded.
SL *name*	Specifies device name or link address.
SP *n*	Sets the number of KBytes of memory peeked on Analyse.
SR	Resets the root transputer and subsystem on the link.
SS	Serves the link, that is, starts up the runtime server environment that enables programs to communicate with the host.
'**SB** *filename*' is equivalent to **SR SS SI SC** *filename*.	

Table 23.1 **iserver** options

23.2.1 Examples of use

UNIX based toolsets:

```
icc hello
ilink hello.tco -f startup.lnk
icollect hello.lku -t
iserver -sb hello.btl -se
```

MS-DOS and VMS based toolsets:

```
icc hello
ilink hello.tco /f startup.lnk
icollect hello.lku /t
iserver /sb hello.btl
/se
```

In this example **iserver** is instructed to load the bootable file **hello.btl** and to terminate on error. The example also shows the steps for compiling, linking and booting the program.

23.2.2 Supplying parameters to the program

Any text supplied on the command line that cannot be interpreted as a server option is passed to the program as a parameter. **iserver** option strings should *not* be used as program parameters.

23.2.3 Checking and clearing the network

On transputer boards the network can be checked and reset using a network check program such as **ispy**.

The **ispy** program is provided as part of the board support software for INMOS *i*q systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program to clear the network is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared. This method is described in section 15.3.4.

23.2.4 Terminating the server

To terminate the server press the ISERVER interrupt key. The **iserver** interrupt key is the same as the standard host system BREAK key.

When the interrupt key is pressed the program does not abort immediately but provides the following options:

```
(x)exit, (s)hell, or (c)ontinue?
```

To confirm your intention to abort the program type '**x**' or press RETURN, which terminates the server.

To suspend the server in order to resume the program later, type 's' for shell.

Note: On some systems the shell option may require a host environment variable. For further information see the Delivery Manual that accompanies the release.

To cancel the interrupt and continue running the program, type 'c'.

23.2.5 Options to use when loading the program

The name of the file containing the program to be loaded is specified using either the 'SC' or the 'SB' option and must be followed by a filename. The 'SB' option has the same effect as specifying the following combination of options: 'SC SI SR SS'.

For programs which communicate via the host file server the 'SS' option *must* be specified in order to start up the host communications environment. When the program has been loaded the server provides runtime access to host services.

To load a program onto a board without resetting the root transputer, use the 'SC' option. This should only be done if the transputer has already been reset, or has a resident program that can interpret the file. To reset the transputer subsystem before loading the program use the 'SR' or 'SB' options.

To terminate the server immediately after loading the program use the 'SR' and 'SC' options together. This combination of options resets the transputer, loads the program onto the board, and terminates.

To load a board in analyse mode, for example when you wish to use the debugger to examine the program's execution, use the 'SA' option to dump the first 8 Kbytes of the transputer's memory (starting from MOSTNEG INT). The data is stored in an internal buffer which is read by the idump tool when programs are to be debugged that use the root transputer.

23.2.6 Specifying a link address – option SL

The server contains a default address or device name for communicating with boot from link boards. The address or name can be changed by specifying the 'SL' option followed by the new value. Addresses can be given as decimal numbers, or in hexadecimal format by prefixing the number with '#'.

The default address is overridden by the value of host environment variable **TRANSPUTER**, if this variable has been set on the system. The address or name defined by this variable is itself overridden by any address or name given after the '**SL**' option.

23.2.7 Terminating on error – option SE

When debugging programs it is useful to force the server to terminate when the subsystem's error flag is set. To do this use the '**SE**' option. The error flag of a transputer is normally set by a program fault.

23.3 Server functions

This section describes the basic set of server functions. All versions of the **iserver** will support these functions, enabling programs to be used with any version of the toolset.

These functions are not intended for applications programmers. They are briefly described here for programmers who wish to implement a server on a new host, or to add new facilities to the existing server. Details of the functions can be found in appendix E.

The functions are divided into three groups:

1 File system commands

2 Host environment commands

3 Server control commands

Commands in each group are summarised in the following tables.

File system commands

Command	Description
Fopen	Opens a file, and returns a stream identifier.
Fclose	Closes a file.
FGetBlock	Reads a block of data, in bytes, with status return.
FPutBlock	Writes a block of data, in bytes, with status return.
Fread	Reads a data block, in bytes.
Fwrite	Writes a data block, in bytes.
Fgets	Reads a line from an open stream.
Fputs	Writes a line to an open stream.
Fflush	Flushes an open stream to the destination device.
Fseek	Resets the file position.
Ftell	Returns the current file position.
Feof	Tests for end-of-file.
Ferror	Returns error status of a given stream.
Isatty	Determines if a stream is a terminal.
Remove	Deletes a file.
Rename	Renames a file.

Host environment commands

Command	Description
Getkey	Reads a character from the keyboard.
Pollkey	Polls the keyboard.
Getenv	Retrieves a host environment variable.
Time	Returns local and universal time.
System	Runs a command on the host system.

Server control commands

Command	Description
Exit	Terminates the server.
CommandLine	Retrieves the server invocation command line.
Core	Retrieves the contents of a peeked transputer's memory.
Version	Retrieves revision data about the server.
MSDOS	Performs an MS-DOS specific operation.

23.4 Error messages

Aborted by user

This message is displayed when the program is interrupted by pressing the BREAK key (Ctrl-C or Ctrl-Break).

Bad link specification

The link name is invalid.

Boot filename is too long, maximum size is *number* **characters**

The specified filename was too long. *number* is the maximum size for filenames.

Cannot find boot file *filename*

The server cannot open the specified file.

Command line too long (at *string***)**

The maximum permissible command line length has been exceeded. The overflow occurred at *string*.

Copy filename is too long, maximum size is *number* **characters**

The specified filename was too long. *number* is the maximum size for filenames.

Error flag raised by transputer

The program has set the error flag on the transputer. Use `idebug` to debug the program.

Expected a filename after -SB option

The '`SB`' option requires the name of a file to load.

Expected a filename after -SC option

The '`SC`' option requires the name of a file to load.

Expected a name after -SL option

The '`SL`' option requires a link name or address.

Expected a number after -SP option

The '`SP`' option requires the number of Kbytes to peek.

Failed to allocate CoreDump buffer

The server was unable to allocate enough memory to copy the required amount of transputer memory.

Failed to analyse root transputer

The link driver could not analyse the transputer.

Failed to reset root transputer

The link driver could not reset the transputer.

Link name is too long, maximum size is *number* **characters**

The specified name was too long. *number* is the maximum length.

Protocol error, *message*

Incorrect protocol on the link. This can happen if there is a hardware fault, or if an incorrect version of the server is used.

message can be any of the following:

got *number* **bytes at start of a transaction**
packet size is too large
read nonsense from the link
timed out getting a further *dataname*
timed out sending reply message

For more information about server protocols see appendix E.

Reset and analyse are incompatible

Reset and analyse options cannot be used together.

Timed out peeking word *number*

The server was unable to analyse the transputer.

Transputer error flag has been set

The program has set the error flag. Debug the program.

Unable to access a transputer

The server was unable to gain access to a link. This occurs when the link address or device name, specified either with the SL option or the `TRANSPUTER` environment variable, is incorrect.

Unable to free transputer link

The server was unable to free the link resource because of a host error. The reason for the error will be host dependent.

Unable to get request from link

The server failed to get a packet from the transputer. This error indicates some general failure.

Unable to write byte *number* **to the boot link**

The transputer did not accept the file for loading. This can occur if the transputer was not reset or because the file was corrupted or in incorrect format.

24 `isim` – T425 simulator

This chapter describes the T425 simulator tool **`isim`** that allows programs to be run and tested without hardware. The chapter explains how to invoke the tool and describes the simulator commands that allow the simulated program to be debugged interactively.

24.1 Introduction

The simulator can run any transputer program that would run on a single IMS T425 mounted on a normal transputer evaluation board. No transputer hardware is required.

Because the simulator runs the same code that would be loaded onto a real transputer, any program that runs satisfactorily in the simulator can be guaranteed to run on an IMS T425. Because all 32-bit transputers are compatible at the source level, the same program can also be run on any IMS 32-bit processor after recompiling for the correct processor type.

The simulator also provides a reduced set of debugging facilities similar to those of the debugger Monitor page. Additional features provided by the simulator are the ability to set break points at transputer addresses and to single step the program.

The simulator can also be used to familiarise new users with transputers and transputer programming and as a teaching aid.

24.2 Running the simulator

To run the simulator use the following command line:

`isim` *program programparameters* {*options*}

where: *program* is the program bootable file.

programparameters is a list of parameters to the program. The list of parameters must follow the bootable filename and parameters must be separated by spaces.

options is a list of **`isim`** options from Table 24.1.

> Options must be preceded by '–' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options may be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
B	Batch mode operation.
BQ	Batch Quiet mode. No progress information is displayed.
BV	Batch Verify mode.
I	Displays full progress information as the simulator runs.
L	Loads the tool onto the transputer board and terminates.
XM	Directs the transputer-hosted versions of the tool to be executed so that they can be restarted without rebooting by the server.
XO	Directs the transputer-hosted versions of the tool to be executed once on the transputer board and then terminate.

Table 24.1 isim options

24.2.1 Example of use

```
isim hello.btl
```

This invokes the simulator on the "Hello World" program.

When first invoked simulator enters the debugging environment. To start the program invoke the 'G' command. The program runs until it completes successfully, a runtime error occurs, or a break point is reached.

If an error occurs the processor halts, the error flag is set, and the program can be debugged using the commands as you would in the debugger Monitor page environment. Typing '?' displays a summary of the commands.

24.2.2 ITERM file

Like the debugger, the simulator reads the ITERM file to determine how to control the terminal screen and to map a few simulator commands. The ITERM file must be defined in the host environment variable **ITERM**.

24.3 Monitor page display

The simulator Monitor page is similar to that of the debugger, which is described in chapter 15. Data displayed at the simulator Monitor page includes the following:

Iptr	Contents of instruction pointer (address of the *next* instruction to be executed).
Wdesc	Contents of workspace descriptor.
Error	Status of error flag.
Halt On Error	Status of halt on error flag.
Fptr1	Pointer to the front of the low priority active process queue. If 'jump 0' breaks are enabled the letter B is displayed after the pointer value.
Bptr1	Pointer to the back of the low priority active process queue.
Fptr0	Pointer to the front of the high priority active process queue.
Bptr0	Pointer to the back of the high priority active process queue.
TPtr1	Pointer to the low priority timer queue. If the timer is disabled the letter X is displayed after the pointer value.
TPtr0	Pointer to the high priority timer queue.

When the simulator is first invoked the Monitor page also displays a memory map of the program.

If **Wdesc** contains the most negative address value, it will be described as 'invalid'. This normally means that no process is executing in the simulator (for example, the program may have become deadlocked). If **Wdesc** contains the address of '**Memstart**' it is displayed as such. An asterisk displayed next to the **Iptr** or **Wdesc** pointer values indicates invalid object code. Invalid pointers may be generated when processes become deadlocked.

The Monitor page also displays the last instruction executed, a summary of Monitor page commands, and, if an error has occurred, the cause of the error.

24.4 Simulator commands

All simulator commands are given at the Monitor page. Many of the commands are similar to those of the debugger Monitor page; for full descriptions of the commands see chapter 15.

24.4.1 Specifying numerical parameters

Some simulator commands require numerical parameters, such as addresses. These can be specified as simple expressions in decimal or hexadecimal format. Expressions can be the sum of two expressions, the result of subtracting one expression from another, or constants. Constants that can be specified: **Areg**, **Breg**, **Creg**, **Iptr**, **Wptr**, decimal constants, hexadecimal constants, or abbreviated hexadecimal constants. Abbreviated hex constants can be created by prefixing the sequence of hex digits with '**%**' or '**#**, which assumes the hexadecimal prefix '**8000....**'. For example, the hex number '**8000F8A**' can be specified in the abbreviated form '**%F8A**'.

24.4.2 Commands mapped by ITERM

Several commands for controlling the display are mapped to specific keys by the ITERM file. The keys to use for these commands can be found by consulting the keyboard layouts supplied in the Delivery Manual.

Simulator debugging commands are listed in the following tables.

Key	Meaning	Description
A	ASCII	Displays a portion of memory in ASCII.
B	Break points	Breakpoint menu.
D	Disassemble	Displays transputer instructions at a specified area of memory.
G	Go	Runs (or resumes) the program.
H	Hex	Displays a portion of memory in hexadecimal.
I	Inspect	Displays a portion of memory in any occam type.
J	Jump into program	Runs (or resumes) the program. Same as **G**.
L	Links	Displays `Iptr` and `Wdesc` for processes waiting for input or output on a link, or for a signal on the **Event** pin.
M	Memory map	Displays a memory map of the simulated transputer.
N	Create dump file	Creates a network core dump file.
P	Program boot	Simulates a program 'boot' onto the transputer.
Q	Quit	Quits the simulator.
R	Run queue	Displays `Iptr` and `Wdesc` for processes on the high or low priority active process queues.
S	Single step	Executes the next transputer instruction.
T	Timer queue	Displays `Iptr` and `Wdesc` and wake-up times for processes on the high or low priority timer queues.
U	Assign register	Assigns a value to a register.
?	Help	Displays help information.

Key	Meaning	Description
[HELP] ♯	Help	Displays help information.
[REFRESH] ♯	Refresh	Redraws the screen.
[FINISH] ♯	Quit	Quits the simulator.
[↑] [↓]		Scrolls the current display.
♯ For key bindings see Delivery Manual. See also section 24.4.2.		

A – ASCII

Displays a segment of memory in ASCII format.

B – Breakpoints

Sets, displays, and cancels break points at specified memory locations or procedure calls. The command displays the Breakpoint Options Page:

```
Breakpoint Options Page

1) Set breakpoint at Address

2) Display breakpoints

3) Cancel breakpoint at Address

Select Option?
```

D – Disassemble

Displays a segment of memory as transputer instructions.

G – Go

Starts the program, or restarts the program after it has been halted (unless the error flag has been set, in which case the program can no longer be run). The program will run until it completes successfully, sets the error flag, or reaches a break point.

To start the program, specify a break point address after the following prompt and press RETURN:

```
(break point address)
```

The default is not to set a break point.

H – Hex

Displays a segment of memory in hexadecimal format.

I – Inspect

Displays a portion of memory in any occam type. See debugger.

J – Jump into program

Same as G – starts or resumes the program.

L – Links

Displays information about links.

M – Memory map

Displays a complete memory map of the program.

N – Create dump file

Creates a network core dump file from which the program can be debugged off-line. The name of the file and the number of bytes to write must be specified. A file extension is not required and should not be specified. The dump file is automatically given the `.dmp` extension.

P – Program boot

Loads the program into transputer memory ('boots the program') so that debugging can start at beginning of the application program without stepping through bootstrap loading code.

Q – Quit

Quits the simulator, and returns to the host operating system.

R – Run queue

Displays the addresses of process waiting on the active process queues.

T – Timer queue

Displays the addresses of process waiting on the timer queues.

U – Assign

Assigns a value to a register. To assign a value, specify the register by name (abbreviations are permitted), and give a value to be assigned to the register.

[?] – Help

Lists the available simulator commands.

[HELP] – Help

Lists the available simulator commands.

[REFRESH] – Refresh

Refreshes the screen.

[FINISH] – Quit

Quits the simulator, and returns to the host operating system.

[↑], [↓], [PAGE UP], and [PAGE DOWN] keys may be used to scroll the display.

24.5 Batch mode operation

isim can be run in batch mode by setting up the environment variable **ISIMBATCH**. If this variable is defined on the system **isim** automatically selects batch mode operation.

24.5.1 Setting up ISIMBATCH

ISIMBATCH is set up on the system as an environment variable using the appropriate command for your host system.

VERIFY and NOVERIFY modes which enable and disable the output of input commands and user responses are defined by setting a value for **ISIMBATCH**. In MS-DOS the command to use is the **set** command. For example:

```
C:\ set ISIMBATCH=VERIFY

C:\ set ISIMBATCH=NOVERIFY
```

In UNIX the equivalent command is **setenv** and on VMS systems the command to use is **define**. Details of how to use these commands can be found in the user documentation for your system.

24.5.2 Input command files

In batch mode **isim** is driven from a command script containing simulator commands and responses to prompts. All prompts by **isim** must be followed by a valid response.

24.5.3 Output

Output can be written to a log file or displayed at the terminal. Input and output streams can be assigned to files or the user's terminal by commands on the host.

isim can be set up to operate in VERIFY or NOVERIFY mode by setting a different values for **ISIMBATCH**. In VERIFY mode all prompts and user responses are included in the output.

24.5.4 Batch mode commands

Batch mode simulator commands '**A**' through '**U**' are the same as the interactive commands. Two additional commands generate special batch mode output:

Key	Meaning	Description
?	Query state	Displays values of registers and queue pointers.
.	Where	Displays next **Iptr** and transputer instruction.

? – Query state

Displays information about the processor state, including current values of registers, queue pointers, and error flag status. For example:

```
Processor state
Iptr           #80000070
Wdesc          #800000C8
Areg           #80000070
Breg           #800000C8
Creg           #80000010
Error          Clear
Halt on Error  Set
Fptr1 (Low     #00000000
Bptr1  queue)  #00000000
Fptr0 (High    #00000000
Bptr0  queue)  #00000000
Tptr1 (timer   #2D2D2D2D
```

```
Tptr0 (queues #2D2D2D2D
```

[.] – Where

Displays the **Iptr** of the next instruction to execute and a disassembly of that instruction. For example:

```
Iptr #80000070.  Low Priority, Next Instruction :
ajw 42 - #2A
```

24.6 Error messages

Cannot open bootfile '*filename***'**

The file containing the code to be run could not be opened or could not be found.

Environment variable 'IBOARDSIZE' does not exist

Board memory size must be specified to the system using the the host environment variable **IBOARDSIZE**. Details of how to set up **IBOARDSIZE** on your system can be found in the Delivery Manual.

Environment variable 'ITERM' not set up

The ITERM definition file for the simulator function keys must be specified in the **ITERM** host environment variable.

IBOARDSIZE is too small (at least *number* bytes required)

The simulator requires a minimum memory size in order to run correctly. Modify the **IBOARDSIZE** variable and retry the program.

ITERM error
Iterm initialisation has failed

The ITERM file for setting up the terminal codes is invalid. *ITERM error* describes the fault in the file.

25 `iskip` – skip loader tool

This chapter describes the skip loader tool that allows programs to be loaded onto transputer networks over the root transputer. The tool sets up a data transfer protocol on the root transputer that allows programs running on the rest of the network to communicate directly with the host.

25.1 Introduction

The skip tool **`iskip`** prepares a network to load a program over the root transputer by setting up a route-through process on the root transputer to transfer data from the application program running on the target network to and from the host computer. A subsequent call to **`iserver`** loads the program onto the network connected to the root transputer, but does not use the root transputer as part of the network. The root transputer is in effect rendered transparent to the rest of the network. The route-through process uses a simple protocol that transfers data byte by byte between the program and the host.

After **`iskip`** has been invoked to set up the data link across the root transputer, the program can be loaded down the host link in the normal way using **`iserver`**.

`iskip` can be used to skip any number of processors and load a program into any part of a network.

`iskip` may only be executed on 32 bit transputers.

25.1.1 Uses of the skip tool

The skip tool has two main uses :

1 To allow programs configured for specific arrangements of transputers to be loaded onto the target network without using the root transputer to run the program. The root transputer helps to load the program onto the network and subsequently hosts a relay process which transfers data from the application program to the host.

  Example of boards supplied by INMOS that can be used to skip load programs are the IMS B004 PC add-in board, which contains a single IMS T414 transputer, and the IMS B008 PC motherboard fitted with a TRAM in slot zero to act as the root transputer. Other slots on the motherboard can be used to accommodate the target network.

2 Programs configured for a network that normally incorporates the root transputer can be debugged without having to use **idump** to save root transputer's memory to disk. Programs can be loaded into the network connected to the root transputer and the debugger can safely run on the root transputer without overwriting the program. The external network must have the correct number and arrangement of processors for the program to be loaded.

This can make debugging transputer programs easier when an extra transputer is available.

25.2 Running the skip tool

To invoke the **iskip** tool use the following command line:

▶ **iskip** *linknumber* {*options*}

where: *linknumber* is the link on the root transputer to which the target transputer network is connected.

options is a list, in any order, of one or more options from table 25.1.

> Options must be preceded by '-' for UNIX based toolsets and '/' for MS-DOS and VMS based toolsets.
>
> Options can be entered in upper or lower case and can be given in any order on the command line.
>
> Options must be separated by spaces.

If no arguments are given on the command line a help page is displayed giving the command syntax.

Option	Description
E	Directs **iskip** to monitor the subsystem error status and terminates when it becomes set.
R	Reset subsystem. Resets all transputers connected downstream of link *linknumber*. Does *not* reset the root transputer.
I	Displays detailed progress information as the tool loads.

Table 25.1 **iskip** options

25.2.1 Examples of use

```
iskip 2 -r                    (UNIX based toolsets)
iskip 2 /r                    (MS-DOS and VAX based toolsets)
```

In this example **iskip** is invoked for a network where the sub-network is wired *down* (see section 15.4.1). The network is prepared to load the program over the root transputer, which is connected to the network via link 2; the '**r**' option resets the target network.

```
iskip 2 -r -e                 (UNIX based toolsets)
iskip 2 /r /e                 (MS-DOS and VAX based toolsets)
```

In this example **iskip** is invoked for a network where the sub-network is wired *subs* (see section 15.4.1). The network is prepared to load the program over the root transputer, which is connected to the network via link 2; the '**r**' and '**e**' options respectively reset the target network and direct **iskip** to monitor the subsystem error status.

25.2.2 Monitoring the error status – option E

The **iskip** '**E**' option should only be used when the sub-network is connected to the Subsystem port of the root transputer i.e. 'wired *subs*'. When the sub-network is connected to the Down port on the root transputer i.e. 'wired *Down*', the '**E**' option must not be used. (For further information about subsystem wiring see section 7.4).

The '**E**' option instructs the server to monitor the subsystem error status and terminate when it becomes set. When it terminates it sets its own error flag in order that the server may detect an error in the subsystem has occured. This allows the program to be debugged.

If the subsystem error status is not properly monitored when the program is run, the server may become suspended when a program error occurs. In these circumstances the server can be terminated using the host system BREAK key.

Note: There is a delay of one second after **iskip** is invoked with the '**E**' option, before monitoring of the subsystem error status begins; if the program fails before this the server may not terminate correctly and the host system BREAK key should be used.

25.2.3 Loading a program

Once **iskip** has been invoked to prepare the network, the program is loaded by invoking **iserver** with the '**SE**', '**SS**' and '**SC**' options. **iserver** must be invoked with the '**SE**' option if the error flag is required to be monitored. This applies whether the **iskip** '**E**' option is used or not. For example:

```
iserver -se -ss -sc myprog.btl  (UNIX based toolsets)
iserver /se /ss /sc myprog.btl  (MS-DOS/VMS based toolsets)
```

Note: After using the skip tool the root transputer must *not* be reset or analysed, that is, **iserver** must *not* be invoked with the '**SR**', '**SB**', or '**SA**' options, while **iskip** is required to run.

25.2.4 Clearing the error flag

If either **iskip** or **iserver** detect that the error flag is set immediately a program starts executing it is likely that the network consists of more processors than are currently being used and that one or more of the unused processors has its error flag set.

On transputer boards the network may be reset using programs such as **ispy** which clear all error flags.

The **ispy** program is provided as part of the board support software for INMOS *i***q** systems products. These products are available separately through your local INMOS distributor.

An alternative to using a network check program to clear the network is to load a dummy process onto each processor. In the act of loading the process code the error flag is cleared. This method is described in section 15.3.4.

25.3 Error messages

This section lists error messages that can be generated by the skip tool.

Called incorrectly

Command line error. Check command line syntax and retry.

Cannot read server's command line

Syntax error. Retry the command.

Duplicate option: *option*

option was supplied more than once on the command line.

No filename supplied

No filename was supplied on the command line.

This option must be followed by a parameter: *option*

The option specified requires a parameter. Check syntax and retry.

Unknown option: *option*

The specified option is invalid. Check option list and retry.

You must specify a link number (0 to 3)

A link number is required. Specify the number of the root transputer link to which the network is connected. If you specify the host link an error is reported.

Appendices

A Toolset standards and conventions

The toolset conforms to a number of conventions for command line syntax, file names, directory searching, and error reporting.

A.1 Command line syntax

All tools in the toolset conform to a common set of conventions for command line syntax.

A.1.1 General conventions

- Commands, and their parameters and options, obey host system standards.
- Filenames, either directly specified on the command line or as arguments to options, must conform to the host system naming conventions.
- Options must be prefixed with the standard option prefix character for the operating system ('-' for UNIX based toolsets and '/' for VMS and MS-DOS based toolsets).
- Command line parameters and options can be specified in any order but must be separated by spaces.
- Lists of arguments to options, where allowed, must be enclosed in parentheses, and the items in the list must be separated by commas.
- If no parameters or options are specified the tool displays a help page that explains the command syntax.

A.1.2 Standard options

Options listed in the following table have the same effect for all tools that use them.

F	Specifies an indirect file (command script).
I	Displays progress data in full.
L	Loads the tool onto a transputer board and awaits a command line. Only applies to transputer hosted tools.
O	Specifies an output file.
XM	Invokes the tool in continuous execution mode. Only applies to transputer hosted tools. Once the tool has completed its current operation it remains resident on the transputer board and can be reinvoked without rebooting onto the transputer board by the server.
XO	Invokes the tool in single invocation mode. Only applies to transputer hosted tools. The tool terminates after execution and has to be rebooted onto the transputer board when it is next invoked. A single invocation is the default.

A.2 Filenames

File names generally follow the naming and character set conventions of the host operating system except that the following directory separator characters cannot be used: colon '**:**', forward slash '**/**', backslash '\', and closing square bracket '**]**'.

A.3 Search paths

The tools locate files by searching a specified *directory path* on the host system. The path is specified using the host environment variable **ISEARCH**.

The tools conform to the following search rules:

1 If the filename contains a directory specification then the filename is used as given. Relative directory names are treated as relative to the directory in which the tool is invoked.

2 If no directory is specified the directory in which the tool is invoked is assumed.

3 If the file is not present in the current directory, the path specified by the environment variable (or logical name) **ISEARCH** is searched. If there are several files of the same name on this path, the first occurrence is used.

4 If the file is not found using the above rules, then the file is assumed to be absent, and an error is reported.

If no search path has been set up then only rules 1 and 2 apply.

All files are written to the current directory.

A.4 Standard file extensions

The toolset uses a standard set of file extensions for source and object files. These extensions are assumed for input files, and created for output files, unless otherwise specified on the command line.

A separate set of extensions for object files must be used where **imakef** is used to build programs for mixed processor networks. These are described separately in section A.5.

A.4.1 'Main path' source and object files

.c C source files.

.tco Compiled binary module produced by the compiler in TCOFF format. Used as input to **ilink** and **ilibr**. Also read by **idebug**.

.lku Linked unit. Created by **ilink** as an executable process with no external references. Used as input to **icollect** (single transputer programs) or within a configuration description. Also read by **idebug**.

.btl Bootable file which can be loaded onto a transputer or transputer network. Created by **icollect** directly from a **.lku** file (single transputer programs) or from a **.cfs** file Bootable files can be sent down a link by **iserver** for immediate execution. Contains information used by **iserver** to control the host link for execution. Also read by **idebug**.

.cfb Configuration binary file containing a description of how code is to be placed on a network, a description of the route to be used to load the network, and the parameters to be passed to each of the processes. Created by **icconf** from a user-defined configuration description and read by **icollect** to prepare a bootable file and by **idebug** to load a network for debugging.

.lib Library file containing a collection of binary modules. Created by **ilibr**.

A.4.2 Other outputs

.btr Executable code without a bootstrap. Created by **icollect** and used as input to **ieprom**.

.rsc Runnable files which can be loaded by a program. These files contain separately compiled units that are designed to be loadable by application programs and executed via special procedures. An application program can determine the various attributes of a linked unit (e.g. workspace required) from the file in order to set up the parameters to call the separately compiled unit.

.hex A hex dump of a file for loading onto a ROM by a custom ROM loader tool.

.ihx Intel hex format files produced by **ieprom** for loading into ROM.

.mot Motorola 'srecord' files produced by **ieprom** for loading into ROM.

A.4.3 Indirect input files

.lnk Linker indirect files which specify the components of a program to be linked. Also used by **imakef** when creating Makefiles.

.lbb Library build files which specify the components of a library to **ilibr**.

A.4.4 Miscellaneous files

Standard extensions are also used for other files supplied with the toolset.

.itm ITERM files containing information about the terminal. Used by tools such as **idebug** to handle the screen in a device-independent manner. Can also be created by users for other terminals. The file is referenced via the environment variable **ITERM**.

.dmp Memory dump and network dump files. Created by **idump** for debugging code on the root transputer (memory dump) or for off-line analysis of a program on a network (network dump). Read by the debugger for post-mortem debugging.

A.5 Extensions required for imakef

The standard file extensions are adequate for simple programs executing on a single transputer, or on a network of transputers all of the same type. If the network is heterogeneous and a particular source file needs to be compiled for more than one transputer type, the following scheme can be used to identify the individual processor types and error modes.

If **imakef** is used to build the program , this scheme *must* be used.

The extended system uses extensions of the form *.fpe*

where: f denotes the type of file and can take the following values:

t for **.tco** equivalents.
l for **.lnk** equivalents.
c for **.lku** equivalents.
r for **.rsc** equivalents.

p denotes the transputer target type or class. This can take the following values:

2 – T212, T222, M212
3 – T225
4 – T414
5 – T425
8 – T800
9 – T801, T805
a – T400, T414, T425, T800, T801, T805
b – T400, T414, T425

e denotes the execution error mode. The values it can take are:

h – on execution, an error will immediately halt the transputer.
s – when an error occurs, the transputer's error flag will be set.
x – the program can be executed in either HALT or STOP mode.

A.6 Error handling

All tools in the toolset display error messages in a standard format. This has certain advantages:

1 The tool generating the error can be identified even when the tool is run out of contact with the terminal.

2 User programs or system utilities can be used to detect and manipulate errors. Some host system editors permit automatic location of errors.

A.6.1 Error displays

Error messages are displayed in a standard format by all tools. The generalised format can be expressed as follows:

severity–toolname–filename (linenumber)–message

where: *severity* indicates the severity level. The four severity categories are described below.

toolname is the standard toolset name for the tool. Names defined using host system abbreviations and batch files are not displayed.

filename and *linenumber* indicate the file and line where the error occurred. They are only displayed if the error occurs in a file. They are commonly displayed when files of the wrong format are specified on the command line, for example, a source file is specified where an object file is expected.

message explains the error and may recommend an action.

A.6.2 Severities

The severity attached to the error indicates the importance of the error to the operation of the tool. It also implies a certain action taken by the tool.

Four severity categories are recognised:

Warning Error Serious Fatal

Warning messages identify minor logical inconsistencies in code, or warn of the impending generation of more serious errors. The tool continues to run and may produce usable output if no serious errors are encountered subsequently.

Error messages indicate errors from which the tool can recover in the short-term but may cause further errors to be generated which may lead to termination. The tool may continue to run but further errors are likely and the tool is likely to abort eventually. No output is produced.

Serious errors are errors from which no recovery is possible. Further processing is abandoned and the tool aborts immediately. No output is produced.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your INMOS field applications engineer.

A.6.3 Runtime errors

Errors which prevent the program from being run are detected by the C runtime system at startup or during program execution. These errors are displayed in a similar format to that used by the tools. All runtime errors are generated at Fatal severity and cause immediate termination of the program. The display format is as follows:

Fatal-C_Library-*reason*

Runtime errors and their meanings are fully described in section 11.6. Errors generated by library functions are also documented under the detailed description of the function.

B Transputer instruction set

This appendix provides a reference for the transputer instruction set as supported by the **__asm** statement. For a detailed specification of each of the instructions available, refer to *'Transputer instruction set: a compiler writer's guide'*.

B.1 Pseudo-instructions

Pseudo-instructions are instructions to the assembler, rather than true transputer instructions.

Expressions used in *load* pseudo instructions must be word sized or smaller, while expressions used in *store* pseudo instructions must be exactly word sized. To load a floating point value, use a *ld* to load its address, then a *fpldnlsn* or *fpldnldb* as required. The following pseudo-instructions are implemented:

align This instruction takes no operands. It generates padding bytes (prefix 0) until the current code address is on a word boundary.

byte This instruction takes as an argument a list of constant values. Only the lower 8 bits of the constant values are generated i.e. if the constant is too large to fit in a byte, only the lower bits will be generated. The assembler copies the literal bytes into the instruction stream.

ld Loads a value into the **Areg**.

ldab Loads values into the **Areg** and **Breg**. The left hand expression is placed in **Areg**.

ldabc Loads values into **Areg**, **Breg** and **Creg**. The leftmost expression is placed in **Areg**.

ldlabeldiff Loads the difference between the addresses of two labels into **Areg**.

st Stores the value from the **Areg**.

stab Stores values from the **Areg** and **Breg**. The leftmost element receives **Areg**.

stabc Stores values from the **Areg**, **Breg**, and **Creg**. The leftmost element receives **Areg**.

word Generates constants of the target-machine word length. This instruction takes as an argument a list of constant values. If the constant is too large to fit in a target-machine word, only the lower bits will be generated.

The *ld*, *ldab*, *st*, and *stab* instructions may use other registers and/or temporaries. *ldabc* and *stabc* may use temporaries.

B.2 `size` option on `__asm` statement

The **`size`** option on **`__asm`** statements that incorporate transputer operations, direct, prefixing and certain pseudo-instructions, forces the instruction to occupy a set number of bytes. If the instruction is shorter than this, it is padded out with trailing prefix 0 instructions. If the instruction cannot fit in the specified number of bytes, a compiler error is reported. The **`size`** option allows instructions to be built with the same size and is intended to assist the creation of jump tables.

B.3 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.

pfix	prefix
nfix	negative prefix

B.4 Direct instructions

The direct instructions form the core of the transputer instruction set. Each direct instruction has a single operand, normally an integer constant, which will be encoded in the instruction itself and, if it is larger than will fit into the 4-bit argument field of the direct instruction, into a series of *pfix* and *nfix* instructions as well.

The transputer architecture is based around a three-register *evaluation stack* and a single base register **Wreg**. The load and store 'local' instructions access a word in memory at a displacement from **Wreg** given by the operand value used. The displacement is scaled by the word size. The load and store 'non-local' instructions use the top evaluation stack register (**Areg**) as the base instead of **Wreg**, allowing computed base addresses to be used.

The operand of the *j*, *cj* and *call* instructions is interpreted as a byte displacement from the instruction pointer (program counter) register **Iptr**. *ldpi* is similar but takes its operand from **Areg**.

adc Add constant operand value to **Areg**

ajw Adjust workspace pointer **Wreg** by constant operand value (scaled by word length)

call Call

cj Conditional jump i.e. 'jump if zero otherwise pop **Areg**'. As with *jump*, a label identifier may be used as argument to this instruction.

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.

ldc Load constant

ldl Load local word

ldlp Load pointer to local word

ldnl Load non-local word

ldnlp Load pointer to non-local word

opr 'operate': the argument to this instruction is a code indicating a zero-operand *indirect* instruction to be executed. Most of the transputer instruction set is made up of these indirect instructions. Normally you would use the mnemonic for the specific indirect instruction which you require: the assembler will encode this as an *opr* instruction on your behalf. However, it is possible to use *opr* explicitly, for example to synthesise the instruction sequence for a new indirect instruction not supported by the T414 and T800 transputers.

stl Store local word

stnl Store non-local word

B.5 Operations

The instructions in this section are all *indirect* instructions built out of the *opr* instruction. None of these instructions take an argument; instead, they work solely with the transputer evaluation stack.

The arithmetic instructions take their operands from the top of the evaluation stack (**Areg**, **Breg**) and push the result value back on the stack in **Areg**.

add	Add
alt	Alt start
altend	Alt end
altwt	Alt wait
and	Bit-wise and
bcnt	Byte count
bsub	Byte subscript (**Areg** = **Areg** + **Breg**)
ccnt1	Check count from 1
clrhalterr	Clear halt-on-error
csngl	Check single
csub0	Check subscript from 0
cword	Check word
diff	Difference
disc	Disable channel
diss	Disable skip
dist	Disable timer
div	Divide
enbc	Enable channel
enbs	Enable skip
enbt	Enable timer
endp	End process
fmul	Fractional multiply (32-bit processors only)
gajw	General adjust workspace
gcall	General call (swap **Areg**↔**Iptr**)
gt	Greater than (result 'true' or 'false', placed in **Areg**)
in	Input message
ladd	Long add
lb	Load byte
ldiff	Long difference
ldiv	Long divide
ldpi	Load pointer to instruction (**Areg** is byte displacement from **Iptr**)
ldpri	Load current priority
ldtimer	Load timer
lend	Loop end
lmul	Long multiply
lshl	Long shift left
lshr	Long shift right
lsub	Long subtract
lsum	Long sum
mint	Minimum integer
move	Move block of memory (src: **Creg** dest: **Breg** len: **Areg**)

mul	Multiply
norm	Normalise
not	Bit-wise not
or	Bit-wise inclusive or
out	Output message
outbyte	Output byte
outword	Output word
prod	Product
rem	Remainder
resetch	Reset channel
ret	Return
rev	Reverse top two stack elements
runp	Run process
saveh	Save high priority queue registers
savel	Save low priority queue registers
sb	Store byte
seterr	Set error
sethalterr	Set halt-on-error
shl	Shift left
shr	Shift right
startp	Start process
sthb	Store high priority back pointer
sthf	Store high priority front pointer
stlb	Store low priority back pointer
stlf	Store high priority back pointer
stoperr	Stop on error
stopp	Stop process
sttimer	Store timer
sub	Subtract
sum	Sum
talt	Timer alt start
taltwt	Timer alt wait
testerr	Test error false and clear
testhalterr	Test halt-on-error
testpranal	Test processor analysing
tin	Timer input
wcnt	Word count
wsub	Word subscript (**Areg** = **Areg** + 4***Breg**)
xdble	Extend to double
xor	Bit-wise exclusive or
xword	Extend to word

B.6 Additional instructions for T400, T414, T425 and TB

The indirect instructions in this section may only be executed on a T400, T414 or T425 processor, although you may use them in *_asm* statements even when compiling for a different processor.

cflerr	Check single-length floating-point infinity or not-a-number
ldinf	Load single-length infinity
postnormsn	Post-normalise correction of single-length floating-point number
roundsn	Round single-length floating-point number
unpacksn	Unpack single-length floating-point number

B.7 Additional instructions for IMS T800, T801 and T805

The instructions in this section may only be executed on T800, T801 and T805 processors, although you may use them in *_asm* statements even when compiling for a different processor.

B.7.1 Floating-point instructions

The indirect instructions in this section provide access to the T8 series built-in floating-point processor. Note that the instructions beginning with '*fpu...*' are doubly indirect: they are accessed by loading an *entry code* constant with a *ldc* instruction, then executing an *fpentry* instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the *fpentry* instruction is also available.

The floating point load and store instructions use the *integer* **Areg** as a pointer to the operand location.

fpadd	Floating-point add
fpb32tor64	Convert 32-bit unsigned integer to 64-bit real
fpchkerr	Check floating error
fpdiv	Floating-point divide
fpdup	Floating duplicate
fpentry	Floating point unit entry: used to synthesise the '*fpu...*' instructions.
fpeq	Floating point equality
fpgt	Floating point greater than
fpi32tor32	Convert 32-bit integer to 32-bit real
fpi32tor64	Convert 32-bit integer to 64-bit real
fpint	Round to floating integer
fpldnladddb	Floating load non-local and add double
fpldnladdsn	Floating load non-local and add single
fpldnldb	Floating load non-local double
fpldnldbi	Floating load non-local indexed double
fpldnlmuldb	Floating load non-local and multiply double
fpldnlmulsn	Floating load non-local and multiply single
fpldnlsn	Floating load non-local single
fpldnlsni	Floating load non-local indexed single
fpldzerodb	Fload zero double
fpldzerosn	Load zero single
fpmul	Floating-point multiply
fpnan	Floating point not-a-number
fpnotfinite	Floating point finite
fpordered	Floating point orderability
fpremfirst	Floating-point remainder first step
fpremstep	Floating-point remainder iteration step
fprev	Floating reverse
fprtoi32	Convert floating to 32-bit integer
fpstnldb	Floating store non-local double
fpstnli32	Store non-local int32
fpstnlsn	Floating store non-local single
fpsub	Floating-point subtract
fptesterr	Test floating error false and clear
fpuabs	Floating-point absolute
fpuchki32	Check in range of 32-bit integer
fpuchki64	Check in range of 64-bit integer
fpuclrerr	Clear floating error
fpudivby2	Divide by 2.0

fpuexpdec32	Divide by 2^{32}
fpuexpinc32	Multiply by 2^{32}
fpumulby2	Multiply by 2.0
fpunoround	Convert 64-bit real to 32-bit real without rounding
fpur32tor64	Convert single to double
fpur64tor32	Convert double to single
fpurm	Set rounding mode to round minus
fpurn	Set rounding mode to round nearest
fpurp	Set rounding mode to round positive
fpurz	Set rounding mode to round zero
fpuseterr	Set floating error
fpusqrtfirst	Floating-point square root first step
fpusqrtlast	Floating-point square root end
fpusqrtstep	Floating-point square root step

B.8 Additional instructions for IMS T225, T400, T425, T800, T801, T805

The indirect instructions in this section supplement the T414's integer instruction set.

bitcnt	Count the number of bits set in a word
bitrevnbits	Reverse bottom n bits in a word
bitrevword	Reverse bits in a word
crcbyte	Calculate CRC on byte
crcword	Calculate Cyclic Redundancy Check (CRC) on word
dup	Duplicate top of stack
wsubdb	Form double-word subscript

The following 2-dimensional block move instructions apply to the **IMS T400, T425, T800, T801 and T805** only:

move2dall	2-dimensional block copy
move2dinit	Initialise data for 2-dimensional block move
move2dnonzero	2-dimensional block copy non-zero bytes
move2dzero	2-dimensional block copy zero bytes

B.9 Additional instructions for the IMS T225, T400, T425, T801 and T805

The indirect instructions listed in this section provide debugging and general support functions.

clrj0break	Clear jump 0 break enable flag
setj0break	Set jump 0 break enable flag
testj0break	Test if jump 0 break flag is set
timerdisableh	Disable high priority timer interrupt
timerdisablel	Disable low priority timer interrupt
timerenableh	Enable high priority timer interrupt
timerenablel	Enable low priority timer interrupt
ldmemstartval	Load value of **MemStart** address
pop	Pop processor stack
lddevid	Load device identity

C Configuration language definition

This appendix defines the syntax of the ANSI C configuration language.

C.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1 Terminal strings of the language – those not built up by rules of the language – are printed in teletype font e.g. **`void`**.

2 Each phrase definition is built up using a double colon and an equals sign to separate the two sides.

3 Alternatives are separated by vertical bars ('|').

4 Optional sequences are enclosed in square brackets ('[' and ']').

5 Items which may be repeated zero or more times appear in braces ('{' and '}').

6 $\{_0$, x $\}$ represents a list of zero or more items of type 'x' separated by commas.

7 $\{_1$, x $\}$ represents a list of one or more items of type 'x' separated by commas.

C.2 Implementation details

1 Subscript ranges for arrays are dependent on the word length of the machine. For 16-bit machines the range is 0 to $2^{15}-1$, for 32-bit machines the range is 0 to $2^{31}-1$.

2 Each line in the source configuration file should not exceed 512 characters, not including leading and following white space.

3 The maximum number of dimensions for a symbol or array constant is 16.

4 The maximum number of characters for an external symbol name in a linked object file is 256.

C.3 Reserved words

The following defines the set of reserved words and predefined attributes and constants that are defined in the ANSI C configuration language.

C.3.1 Keywords

The configuration language's reserved words are as follows:

`by`	`char`	`connect`	`connection`
`define`	`double`	`edge`	`else`
`float`	`for`	`if`	`include`
`input`	`int`	`node`	`on`
`output`	`place`	`rep`	`size`
`to`	`use`	`val`	

C.3.2 Pre-defined attributes

Node attributes

The `element` attribute used for defining the type of a `node` can take the following values:

- `processor` - the node is a processor in a hardware network.
- `process` - the node is a process in a software network.

Note: The names of node attributes are not reserved words and can be freely used as general purpose identifiers by the programmer.

Processor attributes

The attributes defined for nodes of type `processor` are as follows:

- `link` - used by processor and network nodes to define interconnection. Only defined if the `type` attribute has already been defined.
- `type` - used by processor nodes to define processor type. Processor types predefined in standard include files are as follows:

`T400`	`T414`	`T425`	
	`T800`	`T801`	`T805`
	`T212`	`T222`	`T225`

```
M212
```

- **memory** - used by processor nodes to define memory size.

Process attributes

The attribute names currently defined for nodes of type **process** are:

- **stacksize** - used by the process nodes to specify the size of the stack data segment used by the process.
- **heapsize** - used by the process nodes to specify the size of the heap data segment used by the process.
- **priority** - used by process nodes to specify the priority of the process.
- **interface** - used by process nodes to define the type and the default values of parameters to be passed into the process when the process starts executing.
- **order** - used by process nodes to specify the ordering of its code and data segments. The **order** attribute can take the following sub-attributes:

```
code  stack  static  heap  vector
```

C.4 Predefinitions

The following definitions are read from an include file by the configurer at invocation.

C.4.1 Constants

```
val FALSE 0;
val TRUE 1;

val false 0;
val true 1;

val HIGH 0;
val LOW 1;

val high 0;
val low 1;
```

TRUE, **true**, **FALSE**, and **false** are used in expressions where a boolean value is needed.

HIGH, **high**, **LOW**, and **low** can be used to define the execution priority for a process.

C.4.2 Types

```
define node(element = "processor") processor;

define node(element = "process") process;

define processor(type = "T805") t805;
define processor(type = "T801") t801;
define processor(type = "T800") t800;
define processor(type = "T425") t425;
define processor(type = "T414") t414;
define processor(type = "T400") t400;
define processor(type = "T225") t225;
define processor(type = "T222") t222;
define processor(type = "T212") t212;
define processor(type = "M212") m212;

edge host;
```

```
define node(element = "processor") PROCESSOR;

define node(element = "process") PROCESS;

define processor(type = "T805") T805;
define processor(type = "T801") T801;
define processor(type = "T800") T800;
define processor(type = "T425") T425;
define processor(type = "T414") T414;
define processor(type = "T400") T400;
define processor(type = "T225") T225;
define processor(type = "T222") T222;
define processor(type = "T212") T212;
define processor(type = "M212") M212;
```

These definitions are read from the include file **setconf.inc** by the configurer and forms part of its set of predefinitions.

C.5 Language syntax

C.5.1 Configuration

configuration ::= *config-item* {*config-item*}

config-item ::= *declaration*
| *replicator*
| *conditional*
| *directive*

declaration ::= *node-decl*
| *node-attr-decl*
| *nodedef-decl*
| *connect-decl*
| *edge-decl*
| *connector-decl*
| *mapping-decl*
| *numeric-value-decl*
| *compound-decl*
| *use-decl*

compound-decl ::= { *config-item* {*config-item*} }

C.5.2 Language features

letter ::= **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**

digit ::= **0** | **1** | **2** | ... | **9**

id-char ::= *letter* | *digit* | **_**

identifier ::= *letter* {*id-char*}
| **_** {*id-char*}

comment ::= **/*** *any characters except* ***/** *sequence* ***/**

directive ::= **#** *file-include*

file-include ::= **include** *string*

C.5.3 Expressions

octal-digit ::= **0** | **1** | **2** | ... | **7**

hex-digit ::= *digit* | **A** | **B** | ... | **F** | **a** | **b** | ... | **f**

octal ::= **0** *octal-digit* {*octal-digit*}

decimal ::= *digit* {*digit*}

hex ::= **0x** *hex-digit* {*hex-digit*}
| **0X** *hex-digit* {*hex-digit*}

character-const ::= **'** *char* **'**

char ::= *any character except end of line and quote mark*
| *escape-sequence*

escape-sequence ::= **\'** | **\"** | **** | **\?**
| **\a** | **\b** | **\f** | **\n** | **\r** | **\t** | **\v**
| **** *octal-digit [octal-digit] [octal-digit]*
| **\x** {*hex-digit*}

string ::= **"** {*string-char*} **"**

string-char ::= *any character except end of line and double quote mark*
| *escape-sequence*

scale-size ::= **k** | **K** | **l** | **L**

int-const ::= *decimal[scale-size]*
| *octal*
| *hex*

sign ::= **+** | **-**

exponent ::= **E** *[sign] decimal*
| **e** *[sign] decimal*

real-size ::= **f** | **F** | **l** | **L**

real-const ::= *decimal* **.** *[decimal] [exponent] [real-size]*
| *decimal exponent [real-size]*
| **.** *decimal [exponent] [real-size]*

```
array-const   ::= { {1 , exp } }
                | string

subscript     ::= [ exp ] {[ exp ]}

const         ::= int-const
                | real-const
                | character-const
                | array-const [subscript]
numeric-type  ::= int
                | float
                | double
                | char

monadic-op    ::= + | - | ! | ~
                | ( numeric-type )

dyadic-op     ::= + | - | * | / | %
                | & | | | ^ | << | >>
                | && | ||
                | < | > | <= | >= | == | !=

element       ::= identifier {[subscript] . identifier} [subscript]

function-call ::= size ( element )

exp           ::= const
                | element
                | monadic-op exp
                | exp dyadic-op exp
                | exp ? exp : exp
                | ( exp )
                | function-call
```

C.5.4 Replication and conditionals

```
replicator    ::= rep identifier = exp to exp declaration
                | rep identifier = exp for exp declaration

conditional   ::= if exp declaration [else declaration]
```

C.5.5 Numeric value declarations

```
numeric-value-decl ::= val identifier exp ;
```

C.5.6 Network declarations

node-decl ::= *node-type* [({$_1$, *attributes* })] {$_1$, *identifier* [*subscript*] } ;

node-type ::= **node**
| *identifier*

attributes ::= *node-attr*
| *processor-attr*
| *process-attr*
| *identifier* = *exp*

node-attr ::= **element** = *element-type*

element-type ::= **"processor"**
| **"process"**

processor-attr ::= **type** = *processor-type*
| **memory** = *exp*

processor-type ::= **"T212"** | **"T414"** | *etc.*

process-attr ::= **stacksize** = *exp*
| **heapsize** = *exp*
| **priority** = *exp*
| **interface** ({$_0$, *formal-attr* })
| **order** ({$_0$, *order-attr* })

order-attr ::= **code** = *exp*
| **stack** = *exp*
| **static** = *exp*
| **heap** = *exp*
| **vector** = *exp*

formal-attr ::= *formal-type* {$_1$, *identifier* [*subscript*] [= *exp*] }

formal-type ::= *numeric-type*
| **input**
| **output**

node-attr-decl ::= *element* ({$_1$, *attributes* }) ;

nodedef-decl ::= **define** *node-type* [({$_1$, *attributes* })] *identifier* ;

connector-decl ::= **connection** {$_1$, *identifier [subscript]* } ;

connect-decl ::= **connect** *element* , *element* [**by** *identifier [subscript]*] ;
| **connect** *element* **to** *element* [**by** *identifier [subscript]*] ;

edge-decl ::= **edge** {$_1$, *identifier [subscript]* } ;
| **input** {$_1$, *identifier [subscript]* } ;
| **output** {$_1$, *identifier [subscript]* } ;

use-decl ::= **use** *string* **for** *element* ;

C.5.7 Mapping declarations

mapping-decl ::= **place** *element* **on** *element* ;

D Bootstrap loaders

D.1 Introduction

Special loading procedures can be created for the program and used in place of, or in addition to, the standard INMOS bootstrap. The file containing the new bootstrap is specified by invoking the collector with the '**B**' option.

User defined bootstraps must perform all the necessary operations to initialise the transputer, load the network, and set up the software environment for the application program.

Bootstraps are output to the program bootable file as the first section of code in the bootable file. The bootstrap, consisting of the primary and secondary bootstrap sequences, is followed by the standard INMOS network loader program, which is output in small packets, each packet consisting of a maximum of 60 bytes. The last packet of the network loader is followed by a length byte of zero.

In most cases a custom bootstrap will interface directly with the standard INMOS Network Loader, which places various pieces of code and data within the transputer memory in a controlled way. However it is possible to skip the standard loader by sinking its code packets and following the commands used by the network loader that are output after the network loader.

The general format of a custom bootstrap is a concatenated sequence of bootstrap code segments each preceded by a length byte. The sequence can be any length. The bootstrap program must be contained in a single file.

D.1.1 The example bootstrap

The example bootstrap loader provided on the toolset **examples** directory is a combination of several files used in the standard INMOS bootstrap scheme. The files have been combined into a single file to illustrate how to create a user-defined bootstrap; the functionality is the same as that used in the the standard INMOS scheme based on multiple files.

The program is written in transputer code and consists of two parts:

Primary bootstrap – performs processor setup operations such as initialising the transputer links

Secondary bootstrap – sets up the software environment and interfaces to the Network Loader.

Transfer of control

The calling sequence in the standard INMOS scheme is as follows:

The primary loader calls the secondary loader, which then calls the Network Loader. When the Network Loader has completed its work control returns to the secondary loader, which calls the application program via data set up by the Network Loader.

Custom bootstraps should follow the same sequence.

D.1.2 Writing bootstrap loaders

Bootstrap loader programs should be written to perform the same operations as the standard scheme, that is, hardware initialisation, setting up the software environment, and calling the Network Loader. If you skip the Network Loader by sinking its code bytes then you must ensure its function is reproduced in your own code. If you do use the Network Loader you must ensure the interface to it is correct by setting up the invocation stack. The method by which this is achieved can be deduced from the example program listing.

If you wish to make only a few small changes to the standard loader, for example, insert code to initialise some D-to-A convertors, then the example code can be used and the required code can be inserted between the Primary and Secondary Loader code as an additional piece of bootstrap code in the sequence of bootstraps. The rest of the code can be used as it stands.

If you decide to devise your own loading scheme and rewrite the Primary and Secondary Loaders then you should be familiar with the design of the Transputer and its instruction set. For engineering data about the transputer consult the '*Transputer Databook*' and for information about how to use the instruction set see the '*Transputer Instruction Set: a compiler writer's guide*'.

D.2 Example user bootstrap

```
--
-- (c) Inmos 1989
--
-- Assembly file for the Generic Primary bootstrap TA HALT mode
--
--
--
-- VAL   BASE             IS  1 :         -- loop index
-- VAL   COUNT            IS  2 :         -- loop count
--
-- VAL   LOAD_START       IS  0 :         -- start of loader
-- VAL   LOAD_LENGTH      IS  1 :         -- loader block length
-- VAL   NEXT_ADDRESS     IS  2 :         -- start of next block to load
-- VAL   BOOTLINK         IS  3 :         -- link booted from
-- VAL   NEXT_WPTR        IS  4 :         -- work space of loaded code
-- VAL   RETURN_ADDRESS   IS  5 :         -- return address from loader
-- VAL   TEMP_WORKSPACE   IS  RETURN_ADDRESS : -- workspace used by both
--                                        -- preamble and loader
-- VAL   NOTPROCESS       IS  6 :         -- copy of MinInt
-- VAL   LINKS            IS  NOTPROCESS : -- 1st param to loader (MinInt)
-- VAL   BOOTLINK_IN_PARAM  IS  7 :          -- 2nd parameter to loader
-- VAL   BOOTLINK_OUT_PARAM IS  8 :          -- 3nd parameter to loader
-- VAL   MEMORY           IS  9 :         -- 4th parameter to loader
-- VAL   EXTERNAL_ADDRESS IS 10 :          -- 5th parameter to loader
-- VAL   ENTRY_POINT      IS 11 :         -- 6th parameter to loader
-- VAL   DATA_POINT  IS 12 :          -- 7th parameter to loader
-- VAL   ENTRY_ADDRESS    IS 13 :         -- referenced from entry point
-- VAL   DATA_ADDRESS     IS 14 :         -- referneced from Data point
-- VAL   MEMSTART         IS 15 :         -- start of boot part 2
--
--
-- The initial workspace requirement is found by reading the workspace
-- requirement from the loader \occam\ and subtracting the size of the workspace
-- used by both the loader and the bootstrap (\verb|temp.workspace|). This value
-- is incremented by 4 to accommodate the workspace adjustment by the call
-- instruction used to preserve the processor registers.
--
-- initial.adjustment := (loader.workspace + 4) - temp.workspace
-- occam work space, + 4 for call to save registers, - adjustment made
-- when entering occam. Must be at least 4
-- IF
--   initial.adjustment < 4
--     initial.adjustment := 4
--   TRUE
--     SKIP
--
-- set up work space, save registers,
-- save MemStart and NotProcess

    align

    byte   (Endprimary-Primary)  -- Length of the primary bootstrap

Primary:

global Primary

    ajw    INITIAL_ADJUSTMENT -- see above (is 20)
    call   0                  -- save registers

    ldc    _Start - Addr0     -- distance to start byte
    ldpi                      -- address of start
Addr0:
    stl    MEMSTART           -- save for later use

    mint
    stl    NOTPROCESS         -- save for later use
```

```
-- initialise process queues and clear error
    ldl    NOTPROCESS
    stlf                    -- reset low priority queue

    ldl    NOTPROCESS
    sthf                    -- reset high priority queue

-- use clrhalterr here to create bootstrap for REDUCED application

    sethalterr              -- set halt on error
    testerr                 -- read and clear error bit

-- initialise T8 error and rounding
    ldl    MEMSTART         -- Check if processor has floating point unit by
    ldl    NOTPROCESS       --   checking if (memstart >< mint) >= #70
    xor
    ldc    #70              -- Memstart for T5, T8
    rev                     -- B = #70, A = (Memstart >< MINT)
    gt
    eqc    0
    cj     Nofpu

    fptesterr           -- floating check and clear error instruction

Nofpu:

-- initialise link and event words
    ldc    0
    stl    BASE             -- index to words to initialise
    ldc    11               -- no. words to initialise
    stl    COUNT            -- count of words left
Startloop:
    ldl    NOTPROCESS
    ldl    BASE             -- index
    ldl    NOTPROCESS
    wsub                    -- point to next address
    stnl   0                -- put NotProcess into addressed word
    ldlp   BASE             -- address of loop control info
    ldc    Endloop - Startloop -- return jump
    lend                    -- go back if more
Endloop:

-- set up some loader parameters. See the parameter
-- structure of the loader
    ldl    MEMSTART     -- clear data and entry addresses
    stl    DATA_ADDRESS
    ldl    MEMSTART
    stl    ENTRY_ADDRESS

    ldlp   DATA_ADDRESS    -- address of entry word
    stl    DATA_POINT      -- store in param 7

    ldlp   ENTRY_ADDRESS    -- address of entry word
    stl    ENTRY_POINT      -- store in param 6

    ldl    NOT_PROCESS
    stl    EXTERNAL_ADDRESS     -- buffer offset in param 5

    ldl    MEMSTART         -- start of memory
    stl    MEMORY           -- store in param 4

    ldl    BOOTLINK         -- copy of bootlink
    stl    BOOTLINK_IN_PARAM   -- store in param 2

--  Now find the corresponding output link and place in the parameter

    ldl   BOOTLINK
    ldnlp  -4                  -- Calculate the output link address
    stl    BOOTLINK_OUT_PARAM  -- store in param 3
```

```
-- load bootloader over bootstrap
-- code must be 2 bytes shorter than bootstrap
    ldlp   LOAD_LENGTH       -- packet size word
    ldl    BOOTLINK          -- address of link
    ldc    1                 -- bytes to load
    in                       -- input length byte

    ldl    MEMSTART          -- area to load bootloader
    ldl    BOOTLINK          -- address of link
    ldl    LOAD_LENGTH       -- message length
    in                       -- input bootloader

-- enter code just loaded

    pfix   0                 -- For the next bootstrap to be 2 bytes bigger
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0

    ldl    MEMSTART          -- start of loaded code
    gcall                    -- enter bootloader

    align

Endprimary:

--
-- (c) Inmos 1989
-- Assembly file for the generic secondary loader TA IGNORE mode
--
--
--
-- VAL   BASE             IS  1 :         -- loop index
-- VAL   COUNT            IS  2 :         -- loop count
--
-- VAL   LOAD_START       IS  0 :         -- start of loader
-- VAL   LOAD_LENGTH      IS  1 :         -- loader block length
-- VAL   NEXT_ADDRESS     IS  2 :         -- start of next block to load
-- VAL   BOOTLINK         IS  3 :         -- link booted from
-- VAL   NEXT_WPTR        IS  4 :         -- work space of loaded code
-- VAL   RETURN_ADDRESS   IS  5 :         -- return address from loader
-- VAL   TEMP_WORKSPACE   IS  RETURN_ADDRESS : -- workspace used by both
--                                        -- preamble and loader
-- VAL   NOTPROCESS       IS  6 :         -- copy of MinInt
-- VAL   LINKS            IS  NOTPROCESS : -- 1st param to loader (MinInt)
-- VAL   BOOTLINK_IN_PARAM  IS  7 :          -- 2nd parameter to loader
-- VAL   BOOTLINK_OUT_PARAM IS  8 :          -- 3nd parameter to loader
-- VAL   MEMORY           IS  9 :         -- 4th parameter to loader
-- VAL   BUFFER           IS 10 :         -- 5
-- VAL   NEXT_POINT       IS 11 :         -- 6th parameter to loader
-- VAL   ENTRY_POINT      IS 12 :         -- 7th parameter to loader
-- VAL   DATA_POINT       IS 13 :         -- 8th parameter to loader
-- VAL   ENTRY_ADDRESS    IS 14 :         -- referenced from entry point
-- VAL   DATA_ADDRESS     IS 15 :         -- referenced from Data point
-- VAL   NEXT_ADDRESS     IS 16 :         -- referenced from Nexat point
-- VAL   MEMSTART         IS 17 :         -- start of boot part 2
--
--
-- VAL   PACKET_LENGTH    IS 120 :
-- VAL   OCCAM_WORKSPACE  IS 18 :
```

```
    byte   (Endsecondary-Secondary)  -- Length of the secondary boostrap

Secondary:

global Secondary

-- initialise bootloader workspace

    ldc    PACKET_LENGTH     -- buffer size
    ldlp   MEMSTART+1        -- buffer start address
    bsub                     -- end of buffer address
    stl    NEXT_ADDRESS      -- start of area to load loader

    ldl    NEXT_ADDRESS

    ldlp   MEMSTART+1        -- buffer start address
    stl    MEMORY            -- Earliest place to load

    ldlp   TEMP_WORKSPACE    -- pointer to loader's work space zero
    stl    NEXT_WPTR         -- work space pointer of loaded code

    ldc    0
    stl    BUFFER            -- Buffer offset from Buffer start

    ldc    0
    stl    LOAD_LENGTH       -- clear bytes to load

Loadcode:
    ldl    NEXT_ADDRESS      -- address to load loader
    stl    LOAD_START        -- current load point

-- load code until terminator
Startload:
    ldlp   LOAD_LENGTH       -- packet length
    ldl    BOOTLINK          -- address of link
    ldc    1                 -- bytes to load
    in                       -- input length byte

    ldl    LOAD_LENGTH       -- message length
    cj     Endload           -- quit if 0 bytes

    ldl    NEXT_ADDRESS      -- start of area to load loader
    ldl    BOOTLINK          -- address of link
    ldl    LOAD_LENGTH       -- message length
    in                       -- input code block
    ldl    LOAD_LENGTH       -- message length
    ldl    NEXT_ADDRESS      -- area to load
    bsub                     -- new area to load
    stl    NEXT_ADDRESS      -- save area to load

    j      Startload         -- go back for next block
Endload:

-- initialise return address and enter loaded code
    ldc    Return - Addr1    -- offset to return address
    ldpi                     -- return address
Addr1:
    stl    RETURN_ADDRESS    -- save in W0


    ldl    BOOTLINK          -- Get bootlink and save for later
    stl    OCCAM_WORKSPACE   -- Save in area that will not be used
                             -- by network loader

    ldl    NEXT_WPTR         -- wspace of loaded code
    gajw                     -- set up his work space
    ldnl   LOAD_START        -- address of first load packet
    gcall                    -- enter loaded code

Return:
```

```
-- Now set up invocation stack for the Init_system

    ajw    (TEMP_WORKSPACE + 4)-- reset work space after return

    ldl    OCCAM_WORKSPACE     -- get back boot link
    stl    BOOTLINK

    ldl    DATA_ADDRESS     -- get address of processor structure
    ldl    MEMORY
    bsub
    stl    DATA_POINT


    ldl    ENTRY_ADDRESS -- convert to real entry address
    ldl    MEMORY
    bsub
    stl    LOAD_START

    ldl    NOTPROCESS
    stl    NEXT_POINT

    ldl    MEMORY          -- make DATA base offset and CODE base offset the same
    stl    BUFFER          --

    ldl    ENTRY_ADDRESS   --
    stl    TEMP_WORKSPACE  -- Set up entry point

    ldl    NEXT_ADDRESS     -- convert returned address of next sequence to
    ldl    MEMORY           -- a real address
    bsub
    stl    NEXT_ADDRESS

    ldc    0
    stl    LOAD_LENGTH       -- clear bytes to load

    ldlp   NOT_PROCESS      -- Top of temp workspace used by bootloader
    stl    NEXT_WPTR

-- start clock

    ldc    0
    sttimer

    j      Startload        -- Go back for more and over write the network loader


    align

Endsecondary:
```

D.3 The INMOS Network Loader

The following code, written in occam, represents the standard network loader program used by INMOS.

```
--------------------------------------------------------------------------------
--
--
-- This generic loader is written and should be compiled with out any processor type
--
-- dependencies. That is the same object code is used even if the processor is one of
--
-- the sixteen bit variety
--
--
--------------------------------------------------------------------------------
PROC Loader ([4]CHAN OF ANY   links,
             CHAN OF'ANY      bootlink.in, bootlink.out,
             [4]BYTE          memory,
             VAL INT          Buffer.address,
             INT              Next.address,
             INT              Entry.point,
             INT              Data.point)


  --{{{  constants
  VAL   data.field          IS   #3F :
  VAL   data.field.bits     IS   6 :
  VAL   tag.field           IS   #C0 :
  VAL   tag.field.bits      IS   2 :
  VAL   message             IS   0 :
  VAL   number              IS   1 :
  VAL   operate             IS   2 :
  VAL   prefix              IS   3 :
  VAL   tag.prefix          IS   prefix << data.field.bits :
  VAL   message.length      IS   60 :

  VAL   load                IS   0 :
  VAL   pass                IS   1 :
  VAL   open                IS   2 :
  VAL   operate.open        IS   BYTE ((operate << data.field.bits)
                                   \/ open) :
  VAL   close               IS   3 :
  VAL   operate.close       IS   BYTE ((operate << data.field.bits)
                                   \/ close) :
  VAL   address             IS   4 :
  VAL   execute             IS   5 :
  VAL   Data.position       IS   6 :
  VAL   operate.execute     IS   BYTE ((operate << data.field.bits)
                                   \/ execute) :

  VAL   operate.data.postion IS BYTE ((operate << data.field.bits)
                                   \/ Data.position) :
  VAL   code.load           IS   7 :
  VAL   operate.code.load   IS   BYTE ((operate << data.field.bits)
                                   \/ code.load) :

  VAL   code.address        IS   8 :
  VAL   operate.code.address IS BYTE ((operate << data.field.bits)
                                   \/ code.address) :

  VAL  data.load            IS   9 :
  VAL  operate.data.load    IS   BYTE ((operate << data.field.bits)
                                   \/ data.load) :

  VAL  data.address         IS   10 :
  VAL  operate.data.address  IS BYTE ((operate << data.field.bits)
```

```
                                        \/ data.address) :

VAL  Entry.position      IS   11 :
VAL  operate.entry.position IS BYTE ((operate << data.field.bits)
                                        \/ Entry.position) :

VAL  Bootstrap.load      IS   12 :
VAL  Operate.bootstrap.load IS BYTE ((operate << data.field.bits)
                                        \/ Bootstrap.load) :

VAL  Bootstrap.end       IS   13 :
VAL  Operate.bootstrap.end IS BYTE ((operate << data.field.bits)
                                        \/ Bootstrap.end) :

--{{{  VARIABLES
BYTE   command :
INT    Bootstrap.depth, links.to.load, last.address, output.link :
BOOL   loading :
SEQ
  bootlink.in ? command
  WHILE command <> operate.execute
    INT    tag, operand :
    --{{{  process command
    SEQ
      tag := (INT command) >> data.field.bits
      operand := (INT command) /\ data.field
      IF
        --{{{  tag = message
        tag = message
          INT   load.address :
          SEQ
            IF
              --{{{  loading
              loading
                SEQ
                  load.address := last.address
                  last.address := load.address PLUS operand
              --{{{  passing on
              TRUE
                load.address := Buffer.address
            --{{{  read in message
            IF
              operand <> 0
                bootlink.in ? [memory FROM load.address FOR operand]
              TRUE
                SKIP
            --{{{  send message to outputs
            SEQ i = 0 FOR 4
              IF
                (links.to.load /\ (1 << i)) <> 0
                  SEQ
                    links[i] ! command
                    IF
                      operand <> 0
                        links[i] ! [memory FROM load.address FOR operand]
                      TRUE
                        SKIP
                TRUE
                  SKIP
        --{{{  tag = operate
        tag = operate
          IF
            --{{{  operand = load
            operand = load
              SEQ
                loading := TRUE
                links.to.load := 0
            --{{{  operand = data.load
            operand = data.load
              SEQ
```

```
      loading := TRUE
      links.to.load := 0
  --{{{  operand = Code.load
  operand = code.load
    SEQ
      loading := TRUE
      links.to.load := 0
  --{{{  operand = pass
  operand = pass
    SEQ
      loading := FALSE
      links.to.load := 0
  --{{{  operand = open
  operand = open
    INT   depth :
    SEQ
      depth := 1
      WHILE depth <> 0
        SEQ
          bootlink.in ? command
          IF
            command = operate.open
              depth := depth + 1
            command = operate.close
              depth := depth - 1
            TRUE
              SKIP
          IF
            depth <> 0
              links[output.link] !  command
            TRUE
              SKIP
  --{{{  operand = address
  operand = address
    SEQ
      --{{{  read in load offset
      BOOL   more :
      SEQ
        last.address := 0
        more := TRUE
        WHILE more
          SEQ
            last.address := last.address << data.field.bits
            bootlink.in ? command
            last.address := last.address PLUS
                              ((INT command) /\ data.field)

            more := (INT command) >= tag.prefix
      --{{{  entry address
      Next.address := last.address
  operand = Data.position
    SEQ
      --{{{  read in data position offset
      BOOL   more :
      SEQ
        Data.point := 0
        more := TRUE
        WHILE more
          SEQ
            Data.point := Data.point << data.field.bits
            bootlink.in ? command
            Data.point := Data.point PLUS
                              ((INT command) /\ data.field)

            more := (INT command) >= tag.prefix
  operand = Entry.position
    SEQ
      --{{{  read in data position offset
      BOOL   more :
      SEQ
```

```
      Entry.point := 0
      more := TRUE
      WHILE more
        SEQ
          Entry.point := Entry.point << data.field.bits
          bootlink.in ? command
          Entry.point := Entry.point PLUS
                           ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    --{{{  entry address
operand = code.address
  SEQ
    --{{{  read in load offset
    BOOL   more :
    SEQ
      last.address := 0
      more := TRUE
      WHILE more
        SEQ
          last.address := last.address << data.field.bits
          bootlink.in ? command
          last.address := last.address PLUS
                           ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    Entry.point := last.address
operand = data.address
  SEQ
    --{{{  read in load offset
    BOOL   more :
    SEQ
      last.address := 0
      more := TRUE
      WHILE more
        SEQ
          last.address := last.address << data.field.bits
          bootlink.in ? command
          last.address := last.address PLUS
                           ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    --{{{  entry address
    Data.point := last.address
operand = Bootstrap.load
  INT   load.address :
  INT   Bootstrap.length :
  BOOL   more :
  SEQ
    Bootstrap.depth := 0
    Bootstrap.length := 0
    load.address := Buffer.address
    more := TRUE
    bootlink.in ? command
    more := (INT command) >= data.field
    WHILE more
      SEQ
        Bootstrap.depth := Bootstrap.depth PLUS 1
        SEQ i = 0 FOR 4
          IF
            (links.to.load /\ (1 << i)) <> 0
              SEQ
                links[i] ! command
            TRUE
              SKIP
        bootlink.in ? command
        more := (INT command) >= data.field

    operand := (INT command) /\ data.field
```

```
          IF
            Bootstrap.depth > 0
              --{{{  read in message
              SEQ
                IF
                  operand <> 0
                    bootlink.in ? [memory FROM load.address FOR operand]
                  TRUE
                    SKIP
                --{{{  send message to outputs
                SEQ i = 0 FOR 4
                  IF
                    (links.to.load /\ (1 << i)) <> 0
                      SEQ
                        links[i] ! command
                        IF
                          operand <> 0
                            links[i] ! [memory FROM load.address
                                                FOR operand]
                          TRUE
                            SKIP
                    TRUE
                      SKIP
            TRUE
              SEQ
                more := TRUE
                -- The next processor(s) are to be booted !!! --
                -- so build a bootable packet and output down link --
                WHILE more
                  SEQ
                    bootlink.in ? [memory FROM load.address FOR operand]
                    load.address := load.address PLUS operand
                    Bootstrap.length := Bootstrap.length PLUS operand
                    bootlink.in ? command
                    -- Stop building when a proper command
                    -- is received This should be when a
                    -- 'Bootstrap.end' is received
                    more := (INT command) < data.field
                    operand := (INT command) /\ data.field

                SEQ i = 0 FOR 4
                  IF
                    (links.to.load /\ (1 << i)) <> 0
                      SEQ
                        links[i] ! (BYTE Bootstrap.length)
                        IF
                          Bootstrap.length <> 0
                            links[i] ! [memory FROM Buffer.address
                                                FOR Bootstrap.length]
                          TRUE
                            SKIP
                    TRUE
                      SKIP

      operand = Bootstrap.end
        SEQ
          SEQ ii = 0 FOR Bootstrap.depth
            SEQ
              -- Pass on all the other bootstrap ends
              bootlink.in ? command
              SEQ i = 0 FOR 4
                IF
                  (links.to.load /\ (1 << i)) <> 0
                    links[i] ! command
                  TRUE
                    SKIP
          Bootstrap.depth := 0

    --{{{  tag = number
    TRUE
```

```
          SEQ
            output.link := operand
            links.to.load := links.to.load \/ (1 << output.link)

        bootlink.in ? command
:
```

E ISERVER protocol

This appendix describes the protocol of the host file server **iserver** and provides definitions of the functions used to implement it.

E.1 The host file server iserver

The host file server **iserver** is implemented in C using ANSI standard run-time libraries to facilitate porting to other machines. This provides an easy method of porting the toolset (or programs written under the toolset) to new hosts. The server can easily be extended to accommodate a new host, but at the risk of unportability.

The source of the server and of the libraries used to communicate with the server is supplied with the toolset.

E.2 The server protocol

Every communication to and from the server is a packet consisting of a counted array of bytes. The count gives the length of the message and is sent in the first two bytes of the packet as a signed 16 bit number. The structure of a server packet is illustrated in figure E.1.

This protocol has been given the name SP, and is defined in occam as follows:

```
PROTOCOL SP IS INT16::[]BYTE :
```

E.2.1 Packet size

There is a maximum packet size of 512 bytes and a minimum packet size of 8 bytes in the to-server direction (i.e. a minimum message length of 6 bytes). The server may take advantage of this knowledge.

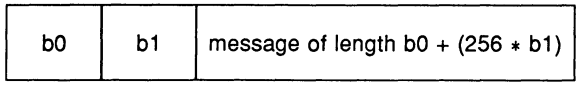


Figure E.1 SP protocol packet

The packet size must always be an even number of bytes. If the number of bytes is odd a dummy byte is added to the end of the packet and the packet byte count rounded up by one.

The hostio library contains routines that ensure that the size restrictions are met when sending a packet to the server (see section E.3).

E.2.2 Protocol operation

Every request sent to the server receives a reply of the same protocol, in strict sequence, and no further requests are accepted until the reply has been sent.

All integer types used by the protocol are signed and are little endian. Numbers are transmitted as sequences of bytes (2 bytes for 16 bit numbers, 4 bytes for 32 bit numbers) with the least significant byte first. Negative integers are represented in 2s complement. Strings and other variable length blocks are introduced by a 16 bit signed count.

All server calls return a result byte as the first item in the return packet. If the operation succeeds the result byte is zero and if the operation fails the result byte is non-zero. The result is one (1) in the special case where the operation fails because the function is not implemented[1]. If the result is non-zero, some or all of the return values may not be present, resulting in a smaller return packet than if the call was successful.

E.3 The server libraries

The i/o library contains all the routines provided in the toolset for communicating with the server. It is implemented via a set of basic routines, hidden from the user, from which the more complex user visible routines are built.

E.4 Porting the server

In order to port the `iserver` to a new machine you must have a C compiler for that machine with ANSI standard libraries. A Makefile that can assist with porting to a new machine is supplied on the toolset 'source' subdirectory.

All the functions described below must be provided by any implementation of `iserver`.

[1]Result values between 2 and 127 are defined to have particular meanings by occam server libraries. Result values of 128 or above are specific to the implementation of a server.

E.5 Server commands

The functions provided by the **iserver** are split into three groups:

1 File commands, for interacting with files

2 Host commands, for interacting with the host

3 Server commands, for interacting with the server itself.

E.5.1 Notation

In the descriptions that follow, the arguments and results of server calls are listed in the order that they appear in the data part of the packet. The size of a packet is the aggregated size of all the items in the packet, rounded up to an even number of bytes.

occam types are used to define data items within the packet. occam types have a clear syntax and are generally self-explanatory but for further details the reader is referred to the '*occam 2 reference manual*', or any good text on occam.

E.5.2 Reserved values

INMOS reserves the following values for its own use:

- Function tags in the range 0 to 127 inclusive.
- Result values in the range 0 to 127 inclusive.
- Stream identifiers 0, 1 and 2.

Some commands may return particular values, which may be reserved. The range of reserved values is given with each command as appropriate.

E.5.3 File commands

Open files are identified with 32 bit descriptors. There are three predefined open files:

0 – standard input
1 – standard output
2 – standard error

If one of these is closed then it may not be reopened.

Fopen – Open a file

```
Synopsis:    StreamId = Fopen( Name, Type, Mode )

To server:    BYTE            Tag = 10
              INT16::[]BYTE   Name
              BYTE            Type = 1 or 2
              BYTE            Mode = 1...6

From server: BYTE             Result
              INT32           StreamId
```

Fopen opens the file **Name** and, if successful, returns a stream identifier **StreamId**.

Type can take one of two possible values:

1 Binary. The file will contain raw binary bytes.

2 Text. The file will be stored as text records. Text files are host-specified.

Mode can have 6 possible values:

1 Open an existing file for input.

2 Create a new file, or truncate an existing one, for output.

3 Create a new file, or append to an existing one, for output.

4 Open an existing file for update (both reading and writing), starting at the beginning of the file.

5 Create a new file, or truncate an existing one, for update.

6 Create a new file, of append to an existing one, for update.

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions, however. An output operation may not follow an input operation without an intervening Fseek, Ftell or Fflush operation.

The number of streams that may be open at one time is host-specified, but will not be less than eight (including the three predefines).

Fclose – Close a file

```
Synopsis:      Fclose( StreamId )

To server:     BYTE            Tag = 11
               INT32           StreamId

From server:   BYTE            Result
```

Fclose closes a stream **StreamId** which should be open for input or output. Fclose flushes any unwritten data and discards any unread buffered input before closing the stream.

Fread – Read a block of data

```
Synopsis:      Data = Fread( StreamId, Count )

To server:     BYTE            Tag = 12
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

Fread reads **Count** bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If **Count** is less than one then no input is performed. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

Result is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

Fwrite – Write a block of data

```
Synopsis:      Written = Fwrite( StreamId, Data )

To server:     BYTE            Tag = 13
               INT32           StreamId
               INT16::[]BYTE   Data

From server:   BYTE            Result
               INT16           Written
```

Fwrite writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of **Data** is less than zero then no output is performed. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position is undefined.

Fwrite returns the number of bytes actually output in **Written**. **Result** is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

If **StreamId** is 1 (standard output) or 2 (standard error) then the write is automatically flushed.

FGetBlock – Read a block of data and return success

```
Synopsis:      Data = FGetBlock( StreamId, Count )

To server:     BYTE            Tag = 23
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

FGetBlock reads **Count** bytes of binary data from the specified stream. Input stops when the specified number of bytes have been read, the end of the file is reached, or an error occurs. If **Count** is less than one (1) no input is performed.

The stream is left positioned immediately after the data read; if an error occurs the position is undefined.

The actual number of bytes read may be less than requested. A **Result** of zero (0) indicates success, any other value failure. If a failure result is returned Feof and Ferror should be used to check for status.

Note: FGetBlock should always be used in preference to Fread, whose function it replaces.

FPutBlock – Write a block of data and return success

```
Synopsis:      FPutBlock( StreamId, String )

To server:     BYTE            Tag = 24
               INT32           StreamId
               INT16::[]BYTE   Data

From server:   BYTE            Result
               INT16           Written
```

FPutBlock writes a given number of bytes of binary data to the specified stream, which should be open for output. If **Data** is less than zero (0) no output occurs. The position of the stream is advanced by the number of bytes actually written successfully. If an error occurs the position is undefined.

The number of bytes actually written is returned in **Written**. The actual number written may be less than requested. A **Result** of zero (0) indicates success, any other value failure. If a failure result is returned Feof and Ferror should be used to check for status.

If **StreamId** is 1 (standard output) the write is automatically flushed.

Note: FPutBlock should always be used in preference to Fwrite, whose function it replaces.

Fgets – Read a line

```
Synopsis:      Data = Fgets( StreamId, Count )

To server:     BYTE            Tag = 14
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

Fgets reads a line from a stream which must be open for input. Characters are read until end of file is reached, a newline character is seen or the number of characters read is not less than `Count`.

If the input is terminated because a newline is seen then the newline sequence is *not* included in the returned array.

If end of file is encountered and nothing has been read from the stream then Fgets fails.

Fputs – Write a line

```
Synopsis:      Fputs( StreamId, String )

To server:     BYTE            Tag = 15
               INT32           StreamId
               INT16::[]BYTE   String

From server:   BYTE            Result
```

Fputs writes a line of text to a stream which must be open for output. The host-specified convention for newline will be appended to the line and output to the file. The maximum line length is host-specified.

Fflush – Flush a stream

```
Synopsis:      Fflush( StreamId )

To server:     BYTE          Tag = 16
               INT32         StreamId

From server:   BYTE          Result
```

Fflush flushes the specified stream, which should be open for output. Any internally buffered data is written to the destination device. The stream remains open.

Fseek – Set position in a file

```
Synopsis:      Fseek( StreamId, Offset, Origin )

To server:     BYTE          Tag = 17
               INT32         StreamId
               INT32         Offset
               INT32         Origin

From server:   BYTE          Result
```

Fseek sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be `Offset` characters from `Origin` which may take one of three values:

1 `Set`, the beginning of the file

2 `Current`, the current position in the file

3 `End`, the end of the file.

For a text stream, `Offset` must be zero or a value returned by Ftell. If the latter is used then `Origin` must be set to 1.

Ftell – Find out position in a file

```
Synopsis:      Position = Ftell( StreamId )

To server:     BYTE            Tag = 18
               INT32           StreamId

From server:   BYTE            Result
               INT32           Position
```

Ftell returns the current file position for `StreamId`.

Feof – Test for end of file

```
Synopsis:      Feof( StreamId )

To server:     BYTE            Tag = 19
               INT32           StreamId

From server:   BYTE            Result
```

Feof succeeds if the end of file indicator for `StreamId` is set.

Ferror – Get file error status

```
Synopsis:      ErrorNo, Message = Ferror(StreamId)

To server:     BYTE            Tag = 20
               INT32           StreamId

From server:   BYTE            Result
               INT32           ErrorNo
               INT16::[]BYTE   Message
```

Ferror succeeds if the error indicator for `StreamId` is set. If it is, Ferror returns a host-defined error number and a (possibly null) message corresponding to the last file error on the specified stream.

Remove – Delete a file

```
Synopsis:      Remove( Name )

To server:     BYTE            Tag = 21
               INT16::[]BYTE   Name

From server:   BYTE            Result
```

Remove deletes the named file.

Rename – Rename a file

```
Synopsis:      Rename( OldName, NewName )

To server:     BYTE            Tag = 22
               INT16::[]BYTE   OldName
               INT16::[]BYTE   NewName

From server:   BYTE            Result
```

Rename changes the name of an existing file `OldName` to `NewName`.

Isatty – Determine if a stream is connected to a terminal

```
Synopsis:      Isatty( StreamId )

To server:     BYTE            Tag = 25
               INT32           StreamId

From server:   BYTE            Result
```

Isatty determines the tty status of the specified stream. It returns success if the stream is connected to a terminal.

E.5.4 Host commands

Getkey – Get a keystroke

```
Synopsis:      Key = GetKey()

To server:     BYTE            Tag = 30

From server:   BYTE            Result
               BYTE            Key
```

GetKey gets a single character from the keyboard. The keystroke is waited on indefinitely and will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

PollKey – Test for a key

```
Synopsis:      Key = PollKey()

To server:     BYTE            Tag = 31

From server:   BYTE            Result
               BYTE            Key
```

PollKey gets a single character from the keyboard. If a keystroke is not available then PollKey returns immediately with a non-zero result. If a keystroke is available it will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

Getenv – Get environment variable

```
Synopsis:      Value = Getenv( Name )

To server:     BYTE            Tag = 32
               INT16::[]BYTE   Name

From server:   BYTE            Result
               INT16::[]BYTE   Value
```

Getenv returns a host-defined environment string for **Name**. If Name is undefined then **Result** will be non-zero.

Time – Get the time of day

```
Synopsis:      LocalTime, UTCTime = Time()

To server:     BYTE            Tag = 33

From server:   BYTE            Result
               INT32           LocalTime
               INT32           UTCTime
```

Time returns the local time and Coordinated Universal Time if it is available. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero.

System – Run a command

```
Synopsis:      Status = System( Command )

To server:     BYTE            Tag = 34
               INT16::[]BYTE   Command

From server:   BYTE            Result
               INT32           Status
```

System passes the string **Command** to the host command processor for execution. If Command is zero length then System will succeed if there is a command processor. If Command is not null then **Status** is the return value of the command, which is host-defined.

E.5.5 Server commands

Exit – Terminate the server

```
Synopsis:      Exit( Status )

To server:     BYTE            Tag = 35
               INT32           Status

From server:   BYTE            Result
```

Exit terminates the server, which exits returning **Status** to its caller.

If **Status** has the special value 999999999 then the server will terminate with a host-specific 'success' result.

If **Status** has the special value -999999999 then the server will terminate with a host-specific 'failure' result.

CommandLine – Retrieve the server command line

```
Synopsis:      String = CommandLine( All )

To server:     BYTE            Tag = 40
               BYTE            All

From server:   BYTE            Result
               INT16::[]BYTE   String
```

CommandLine returns the command line passed to the server on invocation.

If **All** is zero the returned string is the command line, with arguments that the server recognised at startup removed.

If **All** is non-zero then the string returned is the entire command vector as passed to the server on startup, including the name of the server command itself.

Core – Read peeked memory

```
Synopsis      Data = Core( Offset, Length )

To server:    BYTE           Tag = 41
              INT32          Offset
              INT16          Length

From server:  BYTE           Result
              INT16::[]BYTE  Core
```

Core returns the contents of the root transputer's memory, as peeked from the transputer when the server was invoked with the analyse option.

Core fails if **Offset** is larger than the amount of memory peeked from the transputer or if the transputer was not analysed.

If **Offset + Length** is larger than the total amount of memory that was peeked then as many bytes as are available from the given offset are returned.

Version – Find out about the server

```
Synopsis:     Id = Version()

To server:    BYTE           Tag = 42

From server:  BYTE           Result
              BYTE           Version
              BYTE           Host
              BYTE           OS
              BYTE           Board
```

Version returns four bytes containing identification information about the server and the host it is running on.

If any of the bytes has the value 0 then that information is not available.

Version identifies the server version. The byte value should be divided by ten to yield the version number.

Host identifies the host machine and can be any of the following:

1 PC
2 NEC-PC
3 VAX
4 Sun 3
5 IBM 370
6 Sun 4
7 Sun 386i
8 Apollo

OS identifies the host environment and can be any of the following:

1 DOS
2 Helios
3 VMS
4 SunOS
5 CMS

Board identifies the interface board and can be any of the following:

1 B004
2 B008
3 B010
4 B011
5 B014
6 DRX-11
7 QT0
8 B015
9 IBM_CAT
10 B016
11 UDP

Values of **Host**, **OS** and **Board** from 0 to 127, inclusive, are reserved for use by INMOS.

MSDOS – Perform MS-DOS specific function

```
Synopsis:      Id = Version()

To server:     BYTE            Tag = 50

From server:   BYTE            Result
```

F occam interface code

This appendix describes a series of mixed language programming interfaces provided for compatibility with entry points used by previous INMOS C compilers and toolsets.

In previous toolsets mixed language programming with C and occam was achieved using C entry point harnesses and by #IMPORTing pre-linked C programs into occam code. A similar system was used for FORTRAN and Pascal. This form of interfacing has now been replaced by other systems but compatibility has been maintained as far as is possible for C by supplying equivalents of the three entry point types used previously.

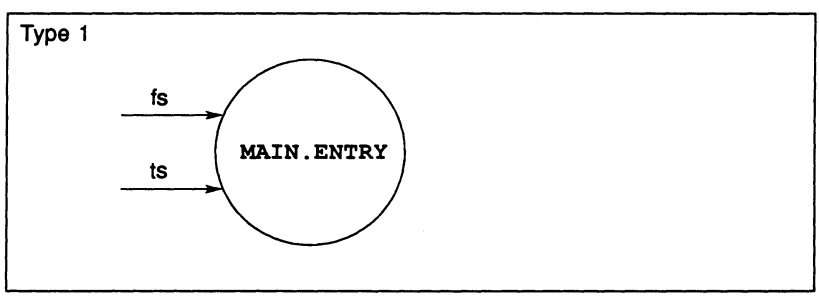
Notes: In the previous system it was possible to bring any number of pre-linked C programs into one occam program using the `#IMPORT` directive. In the current system it is only possible to call one C main program from any single occam program. Multiple importation of programs is supported in the new toolset at configuration level by allowing linked units written in different languages to be incorporated in the same configuration description.

occam code used in this scheme should be linked with the unchecked versions of the occam compiler libraries. This is the default.

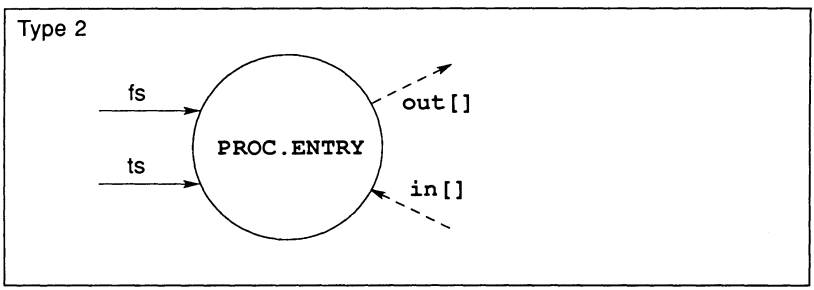
F.1 Interface code

occam interface code provides a fixed interface between occam and C programs. There are three types of interface code, known as types 1, 2, and 3. Descriptions and process diagrams for the three interfaces follow.

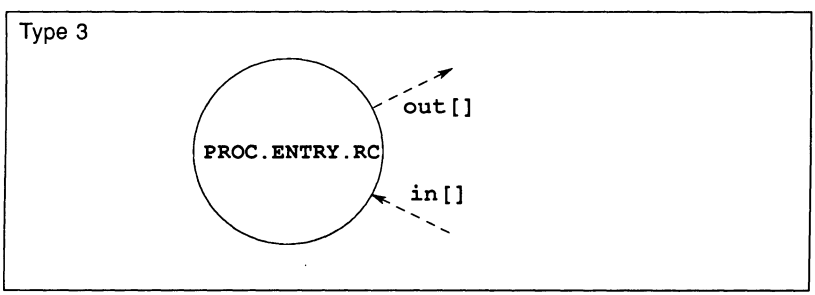
Type 1 : This interface is used when the program runs on a single transputer and communicates only with the host file server.



Type 2 : This interface is used when the program communicates with other processes as well as the host file server. This interface is used with the full version of the C runtime library.



Type 3 : This interface is similar to the type 2 interface except that there is no access to the host file server. The interface is used with the reduced version of the runtime library, which contains only startup, maths, string and channel i/o routines and does not include standard host i/o routines.



Note: In previous toolsets there was a type 3 interface called `PROC.ENTRY.STUB`. This entry point allowed Type 3 processes to be linked with the full runtime library so that functions such as `sprintf` could be used. This is now possible using `PROC.ENTRY.RC` and the stub is no longer required.

Type 2 and 3 interfaces are called from the enclosing occam code and may be a part of a network of occam processes.

F.2 Reserved channels

All equivalent occam processes have four reserved channels, namely `in[0]`, `in[1]`, `out[0]` and `out[1]`. No process which uses host services through the full runtime library should use these channels.

The first two elements of both vectors of channel pointers are reserved as follows:

`out[0]`	Reserved for diagnostic output.
`in[0]`	Reserved for diagnostic input, but not currently used.
`out[1]`	Commands and data from the runtime library to the host file server.
`in[1]`	Responses from the host file server to the runtime library.

F.3 Stack and heap requirements

Data storage (workspace) requirements for C programs are set according to the values of `flag`, `ws1`, and `ws2`. Workspace is allocated by the language compiler and runtime libraries.

Stack, static data and heap requirements vary from program to program, and between languages. The workspace vectors passed to the program must be large enough to accommodate:

- The stack the program needs when it runs.
- All the static data required by the program.
- The heap used by the program and the runtime libraries.

Stack overflow may lead to unpredictable behaviour by the program. For these reasons it is best to run a program, at first, with a large combined stack and heap. Later, when you have run the program and determined stack and heap usage, you may use a separate stack and heap, tailored to your application. Separate workspaces allow you to ensure that the stack is resident in the transputer's internal memory, and enables the program to run faster. Procedures and methods you can use to optimise stacks are described in '*INMOS technical note 17: Performance maximisation*' and '*INMOS technical note 55: Using the occam toolsets with non-occam applications*'.

A minimum stack size of 512 words is recommended.

F.3.1 Stack overflow

Failure or unpredictable behaviour of programs may be due to stack overflow; to test for this in a program, use the procedure outlined below.

1 Initialise the bottom few words of the stack (a falling stack is used) to some pattern of values.

2 Run the program and, after it crashes, use the debugger to examine the values in the stack. If the values you initialised have been changed then stack overflow is likely.

3 Increase the stack size and try again.

The same method can be used to determine static data and heap requirements, except that these use a rising stack.

The following occam fragment gives an example of initialising the bottom of the stack:

```
SEQ i = 0 FOR words.to.initialise
  ws1[i] := i
```

Stack overflow in the C parts of the program can also be detected by using the stack checking mechanism built into the C compiler and libraries.

F.4 Parameters to C `main`

Parameters to C `main` are described by the following function prototype:

```
int main (int argc, char *argv[], char *envp[],
          Channel *in[], int inlen,
          Channel *out[], int outlen);
```

where: **argc** is the number of arguments passed to the program from the environment, including the program name.

***argv** is an array of pointers to those arguments.

***envp** is an array of pointers for the **getenv** library function – implemented in ANSI C as NULL.

Channel *in[] is an array of input arguments.

int inlen is the size of the array.

Channel *out[] is an array of output arguments.

int outlen is the size of the array.

F.5 Type 1 interface

The type 1 interface is used when making a program to run on a single transputer, which does not communicate with any other process apart from the host file server.

C programs that run on a single transputer do not need to use occam. The program should be compiled as usual and then linked with the type 1 occam interface code using **ilink**. This is achieved by linking with the C libraries **centry.lib** and **libc.lib** and specifying **MAIN.ENTRY** as the entry point. This builds the equivalent occam process for the C program, making it appear like occam and enabling you to bootstrap or configure the code in the normal way.

The code for the type 1 process is essentially the same as for any occam process for a single transputer except that an extra parameter is required for the C program's runtime stack (if a separate non-occam stack was requested when **icollect** was invoked). The size of the stack is determined by the parameter supplied with the stack size option '**S**'.

F.5.1 Type 1 procedural interface

The type 1 occam interface is defined as follows:

```
PROC MAIN.ENTRY (CHAN OF SP fs, ts,
                 []INT free.memory,
                 []INT stack.memory)
```

Parameters are described in the following list.

fs	Channel going from the host file server to the program ('from server').
ts	Channel going from the program to the host file server ('to server').
free.memory	Used by the program for its workspace. If the size of the **stack.memory** vector is zero then the **free.memory** vector is used for the program's runtime stack as well as its static and heap data area, otherwise the vector is only used by the program for its static and heap data. This vector represents the amount of free memory left after the program has been loaded. The size of this vector is determined from the environment variable **IBOARDSIZE** which specifies the amount of memory available on the transputer board (in bytes). The value of **IBOARDSIZE** is read at runtime by the bootstrap loader before the program is started.
stack.memory	Used by the program for its runtime stack if the size of the vector is non-zero. The size of this vector is determined when the linked program is made bootable using **icollect** by the parameter supplied with the 'S' option.

F.5.2 Building a type 1 process

The type 1 occam interface code is supplied in centry.lib library.

For example, consider a C program that consists of the following compiled files:

```
main.tco
funcs.tco
```

The program is to run on a T414 transputer.

The program can be linked using one of the following commands:

```
ilink main.tco funcs.tco centry.lib libc.lib
-me MAIN.ENTRY -o cprog.lku              (UNIX based toolsets)

ilink main.tco funcs.tco centry.lib libc.lib
/me MAIN.ENTRY /o cprog.lku        (MS-DOS/VMS based toolsets)
```

When the program has been linked, you can use the collector tool **icollect** in single program mode to produce a bootable program. The 'S' option can be used if required to specify the amount of runtime stack required. If no stack size is requested the freespace is used as a combined heap, stack, and static area.

For the above example, using a specified stack size of 512 words, the collector would be invoked using one of the following commands:

```
icollect cprog.lku -s 512 -t          (UNIX based toolsets)
icollect cprog.lku /s 512 /t(MS-DOS/VMS based toolsets)
```

F.6 Type 2 interface definition

The type 2 interface is used when building a program that will communicate with other processes as well as with the host file server. The program must have been linked with the full version of the runtime library.

F.6.1 Type 2 procedural interface

The type 2 occam interface is defined as follows:

```
PROC PROC.ENTRY (CHAN OF SP fs, ts,
                 VAL INT flag,
                 []INT ws1, ws2,
                 []INT in, out)
```

Parameters are described below.

fs Channel going from the host file server to the program.

ts Channel going from the program to the host file server.

flag Indicates the requirement for one or two workspaces. If the value of **flag** is set to zero then the program will run with two workspace areas, one for static and heap data, the other for the runtime stack. If the value of **flag** is set to one then the program will run with a single combined workspace.

ws1 Used by the program for its workspace. If **flag** is zero then it is used only for the runtime stack; if **flag** is one (1) then it is used as the program's combined workspace.

ws2 Used by the program as its static/heap workspace when **flag** is set to zero. Otherwise unused.

in A vector of pointers to occam channels going to the process.

out A vector of pointers to occam channels going from the process.

Note: The first two elements in the channel pointer vectors **in** and **out** are reserved for use by the C program's runtime system and cannot be used by the program.

F.6.2 Example type 2 wrapping

The following example is of the occam procedure '**call.prog1**', within which a C program is called.

```
PROC call.prog1 (CHAN OF SP fs, ts)

  #INCLUDE "hostio.inc"
  #USE "hostio.lib"
  #USE "centry.lib"      -- C interface code

  VAL flag IS 1 :        -- combined heap and stack
  [100000]INT ws1 :      -- stack and heap for program
  [1]INT ws2 :           -- dummy workspace for
program
  [2]INT in, out :       -- channel pointers

  -- call program
  PROC.ENTRY(fs, ts, flag, ws1, ws2, in, out)
  so.exit(fs, ts, sps.success)
```

:

There are two differences between the code above and that which would have been used with previous toolsets. These differences are as follows:

1. A *library* of C entry points is used in place of a prelinked version of the C program. This is because all linking now takes place together in one pass.

2. The C program is referred to by its interface name **PROC.ENTRY**. Note that this means that only one C program may be called from any occam program.

A program using the above code must be linked with the **centry.lib** and **libc.lib** libraries along with any occam libraries which may be required.

F.7 Type 3 interface definition

The type 3 interface, like the type 2 interface, is used to run programs which communicate with other processes on the same processor or in a network of processes, but which do not require access to host services. Processes built with the type 3 interface can communicate with other processes through channels in the same way as for type 2 processes.

The type 3 interface is used with programs linked with the *reduced* runtime library.

F.7.1 Type 3 procedural interfaces

The interface for type 3 equivalent occam processes is defined below.

```
PROC PROC.ENTRY.RC (VAL INT flag,
                    []INT ws1, ws2,
                    []INT in, out)
```

Parameters are described in the following list.

flag Indicates the requirement one or two workspaces. If the value of **flag** is set to zero then the program will run with two workspace areas, one for static and heap data, the other for the runtime stack. If the value of **flag** is set to one then the program will run with a single combined workspace.

ws1 Used by the program for its workspace. If **flag** is zero then it is used only for the runtime stack; if **flag** is one then it is used as the program's combined workspace.

ws2 Used by the program as its static/heap workspace when **flag** is set to zero. Otherwise it is unused.

in A vector of pointers to occam channels going to the process.

out A vector of pointers to occam channels coming from the process.

Note: The first two elements in the channel pointer vectors **in** and **out** are reserved for use by the C program's runtime system and cannot be used by the occam program.

PROC.ENTRY.RC is supplied in the **centry.lib** library.

F.7.2 Example type 3 wrapping

The following shows how to call an equivalent occam process from occam source, and how to set up the parameters required. The example consists of an occam procedure '**call.prog**' within which a C program is called.

```
PROC call.prog (CHAN OF COMM to.process,
                CHAN OF COMM from.process)

  #USE "centry.lib"      -- C entry point library

  VAL flag IS 0 :        -- separate heap and stack

  [1000]INT ws1 :        -- stack for program
  [40000]INT ws2 :       -- heap for program
  [3]INT in, out :       -- pointers to inputs/outputs

  SEQ

    -- set up user output channel
    LOAD.OUTPUT.CHANNEL(out[2], from.process)

    -- set up user input channel
    LOAD.INPUT.CHANNEL(in[2], to.process)

    -- call program
    PROC.ENTRY.RC(flag, ws1, ws2, in, out)

:
```

Two channels are declared of type COMM, the first being an input channel to the process, the second an output channel from the process. The declaration of protocol type COMM is assumed.

The first statement sets up a pointer to the output channel, using the routine **LOAD.OUTPUT.CHANNEL**. The second statement sets up a pointer to the input channel, using the routine **LOAD.INPUT.CHANNEL**.

As with the type 2 interface there are two differences between the code above and that that would have been used with previous toolsets. The differences are as follows:

1. A *library* of C entry points is used in place of a prelinked version of the C program. This is because all linking now takes place together in one pass.

2. The C program is referred to by its interface name **PROC.ENTRY.RC**. Note that this means that only one C program may be called from any occam program.

A program using the above code must be linked with the **centry.lib** and **libcred.lib** libraries along with any occam libraries which may be required.

G 3L functions supported

G.1 Code compatibility

The ANSI C toolset supports code written using 3L Parallel C at source code level.

G.1.1 Source code

Source code written using the 3L Parallel C toolset must be recompiled before incorporating it into new programs. Errors which indicate non-compliance with ANSI C can be disabled using the '**E-**' series of compiler options.

Other errors should be reviewed as they occur and the source code modified for the new toolset.

G.1.2 Object code

If the source is not available, linked units can be incorporated by converting them to the new TCOFF format using **icvlink**. Any code which is to be converted must be fully linked.

G.2 Parallel functions supported

The ANSI C toolset supports all the routines contained in the 3L Parallel C library packages **thread.h**, **sema.h**, **timer.h**, **chan.h**, and **par.h**, although these header file names should not be used directly. Functions in each package are listed in table G.1.

G.2.1 Header file

3L functions are accessed by including the header file **conndx11.h**. This header file also contains definitions of all constants and literals required by the routines.

G.2.2 Restrictions

All workspace must be allocated from the heap, that is, using the standard memory allocation functions **malloc**, **calloc**, and **realloc**. For example, the **ws** parameter of the **thread_start** functions *must* point to memory taken from the heap.

3L package	Functions supported
thread.h	thread_start thread_create thread_priority thread_deschedule thread_restart thread_stop
sema.h	sema_init sema_signal sema_signal_n sema_wait sema_wait_n
timer.h	timer_after timer_delay timer_now timer_wait
chan.h	chan_init chan_reset chan_in_byte chan_in_byte_t chan_in_word chan_in_word_t chan_in_message chan_in_message_t chan_out_byte chan_out_byte_t chan_out_word chan_out_word_t chan_out_message chan_out_message_t
par.h	par_malloc par_free par_printf par_fprintf

Table G.1 3L functions supported by ANSI C

Code which uses the 3L functions may run slower than equivalent code written using the standard INMOS concurrency functions.

H ITERM

H.1 Introduction

This appendix describes the format of ITERM files; it is included for people who need to write their own ITERM because they are using terminals that are not supported by the standard ITERM file supplied with the toolset. You may of course wish to tailor a standard ITERM to suit your own needs.

ITERMs are ASCII text files that describe the control sequences required to drive terminals. Screen oriented applications that use ITERM files are terminal independent.

ITERM files are similar in function to the UNIX *termcap* database and describe input from, as well as output to, the terminal. They allow applications that use function keys to be terminal independent and configurable.

Within the toolset, the ITERM file is only used by the debugger tool **idebug** and the T425 simulator tool **isim**.

H.2 The structure of an ITERM file

An ITERM file consists of three sections. These are the *host*, *screen* and *keyboard* sections. Sections are introduced by a line beginning with the section letters '**H**', '**S**' or '**K**'. Case is unimportant and the rest of the line is ignored. Sections consist of a number of lines beginning with a digit. A section is terminated by a line beginning with the letter '**E**'. The *host* section must appear first; other sections may appear in any order in the file. Sections must be separated by at least one blank line.

The syntax of the lines that make up the body of a section is best described in an example:

```
3:34,56,23,7.    comments
```

Each line starts with the index number followed by a colon and a list of numbers separated by commas. Each line is terminated by a full stop ('.') and anything following it is treated as a comment. Spaces are not allowed in the data string and an entry cannot be split across more than one line.

Comment lines, beginning with the character '#', may be placed anywhere in an ITERM file. Extra blank lines in the file are ignored.

The index numbers in each section correspond to an agreed meaning for the data. In the following sections the meaning of the data in each of the three sections is described in detail.

H.3 The host definitions

H.3.1 ITERM version

This item identifies an ITERM file by version. It provides some protection against incompatible future upgrades.

e.g. `1:2.`

H.3.2 Screen size

This item allows applications to find out the size of the terminal at startup time. The data items are the number of columns and rows, in that order, available on the current terminal.

e.g. `2:80,25.`

Screen locations should be numbered from 0, 0 by the application. Terminals which use addressing from 1, 1 can be compensated for in the definition of goto X, Y.

H.4 The screen definitions

The lists of values in the screen section represent control codes that perform certain operations; the data values are ASCII codes to send to the display device.

ITERM version 2 defines the indices given in table H.1. These definitions are used in the example ITERM file; for a complete listing of the file see section H.7.

For example, an entry like: '`8:27,91,75.`' indicates that an application should output the ASCII sequence '`ESC [ K`' to the terminal output stream to clear to end of line.

Index	Screen operation	Index	Screen operation
1	cursor up	9	clear to end of screen
2	cursor down	10	insert line
3	cursor left	11	delete line
4	cursor right	12	ring bell
5	goto x y	13	home and clear screen
6	insert character	20	enhance on (not used)
7	delete character at cursor	21	enhance off (not used)
8	clear to end of line		

Table H.1 ITERM screen operations

H.4.1 Goto X Y processing

The entry for 5, 'goto X Y', requires further interpretation by the application. A typical entry for 'goto X Y' might be:

```
5:27,-11,32,-21,32
```

The negative numbers relate to the arguments required for X and Y.

...,-ab,nn,...

where: *a* is the argument number (i.e. 1 for X, 2 for Y).

b controls the data output format.
If *b*=1 output is an ASCII byte (e.g. 33 is output as `!`).
If *b*=2 output is an ASCII number (e.g. 33 is output as `3 3`).

nn is added to the argument before output.

As a complete example, consider the following ITERM entry in the screen section:

```
5:27,91,-22,1,59,-12,1,72. ansi cursor control
```

This would instruct an application wishing to move the terminal cursor to X=14, Y=8 (relative to 0,0) to output the following bytes to the screen:

```
Bytes in decimal: 27   91  57  59  49  53  72
Bytes in ASCII:   ESC  [   9   ;   1   5   H
```

H.5 The keyboard definitions

Each index represents a single keyboard operation. The data specified after each index defines the keystroke associated with that operation. Multiple entries for the same index indicate alternative keystrokes for the operation.

ITERM version 2 defines the indices given in table H.2. These definitions are used in the example ITERM file; for a complete listing of the file see section H.7.

Index	Function	Index	Function
2	delete character	39	goto line
6	cursor up	40	backtrace
7	cursor down	41	inspect
8	cursor left	42	channel
9	cursor right	43	top
12	delete line	44	retrace
14	start of line	45	relocate
15	end of line	46	info
18	line up	47	modify
19	line down	48	resume
20	page up	49	monitor
21	page down	50	word left
26	enter file	51	word right
27	exit file	55	top of file
28	refresh	56	end of file
29	change file	62	toggle hex
31	finish	65	continue from
34	help	66	toggle breakpoint
36	get address	67	search

Table H.2 ITERM key operations

H.6 Setting up the ITERM environment variable

To use an ITERM the application has to find and read the file. An environment variable (or logical name on VMS) called **ITERM** should be set up with the pathname of the file as its value. For example, under MS-DOS the command would be:

```
C:\> set ITERM=C:\ITOOLS\TOOLS\PCBANSI.ITM
```

Under UNIX you would set an environment variable. For example, the command for **csh** users might be:

```
%  setenv ITERM ~/.iterm
```

Under VMS you would define a logical name. For example:

```
$ DEFINE ITERM SYS$LOGIN:VT100.ITM
```

For more details about setting environment variables see the Delivery Manual that accompanies the release.

H.7 An example ITERM

This is the toolset ITERM file for the IBM PC using the ANSI screen driver.

```
#-------------------------------------------------
#
#  IBM PC (BANSI) ITERM data file (derived from TDS3 ITERM)
#  Support for idebug and isim
#  IDEBUG version for BANSI.SYS driver:
#  Special care needed on screen codes 6, 7, 9, 10, 11
#
#  V1.1 - 10 July 90  (NH)  Updated idebug and isim support
#
#-------------------------------------------------

host section
1:2.                          version
2:80,25.                      screen size
end of host section

#  screen control characters

screen section
#                             DEBUGGER        SIMULATOR
```

```
1:27,91,65.                                cursor up
2:27,91,66.                                cursor down
3:27,91,68.                  cursor left   cursor left
4:27,91,67.                                cursor right
5:27,91,-22,1,59,-12,1,72.   goto x y      goto x y
6:27,91,64.                  insert char   insert char
7:27,91,80.                  delete char   delete char
8:27,91,75.                  clear to eol  clear to eol
9:27,91,74.                  clear to eos  clear to eos
10:27,91,76.                 insert line   insert line
11:27,91,77.                 delete line   delete line
12:7.                        bell          bell
13:27,91,50,74.              clear screen  clear screen
end of screen section

keyboard section
#              KEY           DEBUGGER      SIMULATOR
#
2:8.           # BACKSPACE    del char
6:0,72.        # UP           cursor up     cursor up
7:0,80.        # DOWN         cursor down   cursor down
8:0,75.        # LEFT         cursor left   cursor left
9:0,77.        # RIGHT        cursor right  cursor right
12:0,110.      # ALT F7       delete line
12:21.         # CTRL U       delete line
12:24.         # CTRL X       delete line
14:0,65.       # F7           start of line start of line
15:0,66.       # F8           end of line   end of line
18:0,67.       # F9           line up
19:0,68.       # F10          line down
20:0,112.      # ALT F9       page up       page up
21:0,113.      # ALT F10      page down     page down
26:0,71.       # NUM 7        enter file
27:0,73.       # NUM 9        exit file
28:27.         # ESC          refresh       refresh
29:0,87.       # SHIFT F4     change file
31:0,117.      # CTRL NUM 1   finish
34:0,59.       # F1           help          help
36:0,63.       # F5           get address
39:0,64.       # F6           goto line
40:0,129.      # ALT 0        backtrace
41:0,120.      # ALT 1        inspect
42:0,121.      # ALT 2        channel
43:0,122.      # ALT 3        top
44:0,123.      # ALT 4        retrace
45:0,124.      # ALT 5        relocate
46:0,125.      # ALT 6        info
47:0,126.      # ALT 7        modify
48:0,127.      # ALT 8        resume
```

```
49:0,128.       # ALT 9         monitor
50:0,90.        # SHIFT F7      word left
50:6.           # CTRL F        word left
50:0,115.       # CTRL NUM 4    word left
51:0,91.        # SHIFT F8      word right
51:7.           # CTRL G        word right
51:0,116.       # CTRL NUM 6    word right
55:0,92.        # SHIFT F9      top of file
55:20.          # CTRL T        top of file
56:0,93.        # SHIFT F10     end of file
56:2.           # CTRL B        end of file
62:0,108.       # ALT F5        toggle hex
65:0,105.       # ALT F2        continue from
66:0,99.        # CTRL F6       toggle break
67:0,88.        # SHIFT F5      search

end of keyboard stuff

#  idebug key that isn't really part of iterm but its here
all the same !
#
#               INTERRUPT       CTRL A     --   IDEBUG

#  eof THAT'S ALL FOLKS
```

I Glossary

Analyse To assert a signal to a transputer forcing it to halt at the next descheduling point, to allow the state of the processor to be read. In the context of 'analysing a network', to analyse all processors in the network.

Also refers to one of the system control functions on transputers and the pin on which the function is asserted.

Backtrace Within the debugger an simulator tools, to move from a position within a procedure or function body to the call of that procedure or function.

Bootable code Self-starting program code, that can be loaded onto a transputer or transputer network down a transputer link and run. Bootable code is produced by `icollect` from linked units (single transputer programs) or configuration binary files (configured programs).

Bootstrap A transputer program, loaded from a ROM or over a link after the transputer has been reset or analysed, which initialises the processor and loads a program for execution (which may be another loader).

Configuration The association of components of a program with a set of physical resources. Used in this manual to refer to the specific case of allocating software processes to processors in a network, and channels to links between processors. The term is also used, depending on the context, to describe the act of deciding on these allocations for a program, the configuration code which describes such a set of allocations, and the act of applying the configurer to a configuration description.

Configurer The tool which assigns processes and channels on a specified configuration of transputers. The output from the tool is a configuration binary file for input to `icollect`.

Deadlock A state in which one or more concurrent processes can no longer proceed because of a communication interdependency.

Error mode The compilation mode of a program that determines what happens when a program error (such as an array bounds violation) occurs. Programs are compiled by `icc` in UNIVERSAL mode, which is the mode

that can be mixed with HALT and STOP code generated by other INMOS language tooslets.

Error signal In the transputer, an external signal used to indicate that an error has occurred in a running program. Also refers to one of the system control functions on transputers. Error signals can be OR-ed together on transputer boards to indicate an error has occurred in one of the transputers in the network.

Hard channels Channels which are mapped onto links between processors in a transputer network (cf. *Soft channels*).

Host The computer which is running the toolset host file server and providing the filing system and terminal i/o.

Host file server A file server which provides access to the filing system and terminal i/o of a host operating system, which may be used when running standalone programs.

Include file A file containing source code which is incorporated into a program using the **`#include`** directive. Include files are by convention given the **`.h`** extension.

Library A collection of separately compiled procedures or functions, created by the toolset librarian **`ilibr`**, which may be shared between parts of a program or between different programs.

Library build file A file containing a list of input files for the librarian tool **`ilibr`**. Each file forms a separately loadable module in the library. Library build files should have the **`.lbb`** extension.

Link In the context of transputer hardware, the serial communication link between processors. Used as a verb in the context of program compilation, to collect together all the code for a program or compilation unit, resolving all references and recompiling where necessary, and place the collected code into a single file.

Linker The program or tool which links a program or compilation unit.

Loader Depending on the context, refers to the part of the host file server which loads a transputer network or to a small program which is loaded into

a transputer, and which then distributes code to other transputers and loads a larger program on top of itself.

Makefile An input file for a MAKE program. A Makefile contains details of file dependencies and directions for rebuilding the object code. Makefiles are created for the toolset using **`imakef`**.

Network A set of transputers connected together using links as a connected graph, that is, in such a way that there is a path, via links and other transputers, from each transputer to every other transputer in the set.

Newline sequence The sequence of ASCII characters, defined within the host file server, that directs a new line to be started on the terminal display or within a file.

Object code Intermediate code between source and bootable files. Object code cannot be directly loaded onto a transputer and run.

Peek and poke To read and write locations in a transputer's memory, by communication over a link, while the transputer is waiting for a bootstrap.

Preamble The part of a transputer loader program that initialises the state of the processor.

Priority In the transputer, the priority level at which the currently executing process is being run. INMOS transputers support two levels of priority, known as 'high' and 'low'.

Process Self-contained, independently executable code.

Protocol The pattern of communications between two processes, often including communications on more than one channel.

Reset The transputer system initialisation control signal. Also refers to the pin on which the signal is asserted.

Root transputer (or Root processor) The processor in a transputer network which is physically connected to the host computer, and through which the network is loaded or analysed.

Separate compilation A self-contained part of a program may be separately compiled, so that only those parts of a program which have changed since the last compilation need to be recompiled.

Server A program running in the host computer attached to a transputer network, which provides access to the filing system and terminal i/o of the host computer. The server can also be used to load the program onto the network.

Soft channels Channels declared and used within a process running on a single transputer. (cf. *Hard channels*). Soft channels are implemented by a single word in memory.

Standard error The host system error handler. Errors directed to standard error are displayed in a host-defined way, for example, on the terminal screen. For details of how to modify standard error on the system, consult the operating system documentation.

Standard input The host system input handler. Specifies the standard input device, for example the terminal keyboard or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Standard output The host system output handler. Specifies the standard output device, for example, the terminal screen or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Subsystem In transputer board architecture, the combination of the Reset, Analyse and Error signals which allows the board to control another board on its subsystem port.

Target transputer The transputer on which the code is intended to run. The transputer type, or a restricted set of types defined in a transputer class, is defined when the program is compiled, using command line options.

Vector space The data space required for the storage of vectors (arrays) within an occam program.

Workspace The data space required by an occam process; when used in contrast to *Vector space*, refers to the data space required for scalars within the occam process.

J Bibliography

This appendix lists C language reference books and transputer-related publications.

J.1 Reference books

Brian W. Kernighan & Dennis M. Ritchie
The C programming language (First Edition)
Prentice Hall 1978

Brian W. Kernighan & Dennis M. Ritchie
The C programming language (Second Edition – ANSI C)
Prentice Hall 1988

Samuel P. Harbison and Guy L. Steele
C: A Reference Manual (Second Edition – ANSI C)
Prentice Hall 1987

J.2 INMOS publications

INMOS Ltd
The Transputer Databook (Second Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Applications Notebook: Architecture and Software (First Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Applications Notebook: Systems and Performance (First Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Development and iq Systems Databook (First Edition 1989)
INMOS 1989

INMOS Ltd
Transputer instruction set: a compiler writer's guide
Prentice Hall 1988

INMOS Ltd
occam 2 Reference Manual
Prentice Hall 1988

J.3 INMOS technical notes

S Ghee
IMS B004 IBM PC add-in board
Technical note 11
72 TCH 011

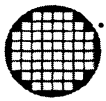
N Miller
Exploring Multiple Transputer Arrays
Technical note 24
72 TCH 024

J M Wilson
Analysing transputer networks
Technical note 33
72 TCH 033

J M Wilson
Loading transputer networks
Technical note 34
72 TCH 034

Index

inmos



Worldwide Headquarters

INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UNITED KINGDOM
Telephone (0454) 616616
Fax (0454) 617910

Worldwide Business Centres

USA

INMOS Business Centre
Headquarters (USA)
SGS-THOMSON Microelectronics Inc.
2225 Executive Circle
PO Box 16000
Colorado Springs
Colorado 80935-6000
Telephone (719) 630 4000
Fax (719) 630 4325

SGS-THOMSON Microelectronics Inc.
Sales and Marketing Headquarters (USA)
1000 East Bell Road
Phoenix
Arizona 85022
Telephone (602) 867 6100
Fax (602) 867 6102

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
Lincoln North
55 Old Bedford Road
Lincoln
Massachusetts 01773
Telephone (617) 259 0300
Fax (617) 259 4420

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
9861 Broken Land Parkway
Suite 320
Columbia
Maryland 21045
Telephone (301) 995 6952
Fax (301) 290 7047

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
200 East Sandpointe
Suite 650
Santa Ana
California 92707
Telephone (714) 957 6018
Fax (714) 957 3281

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
2620 Augustine Drive
Suite 100
Santa Clara
California 95054
Telephone (408) 727 7771
Fax (408) 727 1458

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
1310 Electronics Drive
Carrollton
Texas 75006
Telephone (214) 466 8844
Fax (214) 466 7352

ASIA PACIFIC

Japan

INMOS Business Centre
SGS-THOMSON Microelectronics K.K.
Nisseki Takanawa Building, 4th Floor
18–10 Takanawa 2-chome
Minato-ku
Tokyo 108
Telephone (03) 280 4125
Fax (03) 280 4131

Singapore

INMOS Business Centre
SGS-THOMSON Microelectronics Pte Ltd.
28 Ang Mo Kio Industrial Park 2
Singapore 2056
Telephone (65) 482 14 11
Fax (65) 482 02 40

EUROPE

United Kingdom

INMOS Business Centre
SGS-THOMSON Microelectronics Ltd.
Planar House
Parkway Globe Park
Marlow
Bucks SL7 1YL
Telephone (0628) 890 800
Fax (0628) 890 391

France

INMOS Business Centre
SGS-THOMSON Microelectronics SA
7 Avenue Gallieni
BP 93
94253 Gentilly Cedex
Telephone (1) 47 40 75 75
FAX (1) 47 40 79 27

West Germany

INMOS Business Centre
SGS-THOMSON Microelectronics GmbH
Bretonischer Ring 4
8011 Grasbrunn
Telephone (089) 46 00 60
Fax (089) 46 00 61 40

Italy

INMOS Business Centre
SGS-THOMSON Microelectronics SpA
V.le Milanofiori
Strada 4
Palazzo A/4/A
20090 Assago (MI)
Telephone (2) 89213 1
Fax (2) 8250449