inmos

occam 2
toolset
user manual – part 1

(User guide and tools)

INMOS Limited

Contents overview

Contents

Preface

How to use the manual

User Guide

<table>
<tr><td>1</td><td>Introduction</td><td>Introduces the toolset and transputer programming.</td></tr>
<tr><td>2</td><td>Overview of the toolset</td><td>An overview of the toolset containing brief descriptions of each tool, an introduction to the libraries, and explanations of the toolset conventions.</td></tr>
<tr><td>3</td><td>Getting started</td><td>Shows the command sequences to generate single transputer programs.</td></tr>
<tr><td>4</td><td>Programming single transputers</td><td>An introduction to programming single transputers, with worked examples.</td></tr>
<tr><td>5</td><td>Configuring transputer networks</td><td>An introduction to programming and configuring transputer networks, with examples.</td></tr>
<tr><td>6</td><td>Loading transputer programs</td><td>Describes how to load programs onto transputers and transputer networks, with descriptions of the tools that are used.</td></tr>
<tr><td>7</td><td>Debugging occam programs</td><td>Describes how to use the debugger to debug occam programs in post-mortem and breakpoint modes.</td></tr>
<tr><td>8</td><td>Access to host services</td><td>Describes how to access host services using the host file server and i/o libraries.</td></tr>
<tr><td>9</td><td>Mixed language programming</td><td>Describes how to use C in occam programs.</td></tr>
<tr><td>10</td><td>Low level programming</td><td>Describes the low level facilities of occam 2.</td></tr>
<tr><td>11</td><td>EPROM programming</td><td>Describes the EPROM programming facilities of the toolset.</td></tr>
</table>

Tools

The Index

Contents

Preface

This manual is a combined user and reference guide to the occam 2 toolset.
Part 1 *'User guide and tools'* (this book) describes the toolset and shows how
it is used to develop and run transputer programs. Part 2 *'occam libraries and
appendices'* (72 TDS 276 02) describes the libraries supplied with the toolset
and provides reference data in the form of appendices. A guide to how to use
this manual, follows immediately after this preface.

The occam 2 toolset

The occam 2 toolset is a set of software tools for developing transputer pro-
grams on host systems. Used with the occam libraries, it provides a complete
environment for developing programs on transputers and transputer networks.

The toolset allows occam programs to be written using any convenient text
editor. Programs are then compiled and linked using programs resident on the
host or running on the transputer board. Self-booting code for single transputers
and multitransputer networks is produced using separate tools, and loaded from
the host system down the transputer link.

Tools that assist program development include a librarian tool for building code
libraries, a network debugger which provides both interactive and post-mortem
debugging facilities, and a transputer simulator that allows programs to be tested
without transputer hardware. A Makefile generator is provided to assist with
program version control, and a binary lister tool allows object files to be decoded
and displayed in a readable form.

Transputer programs are normally written in occam to make full use of trans-
puter parallel processing. Programs can also be written in C and included in
occam programs as separately compiled procedures.

The occam 2 toolset is intended for developing programs on transputers and
transputer boards that are loaded from the host via a transputer link. Boards
that boot from on-board ROM require application software to be in a format
suitable for blowing into ROM. Two tools are provided with the toolset to support
EPROM programming, they are the EPROM program formatting tool and the
EPROM memory configurer.

Host versions

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

IMS D7205 – IBM and NEC PC running MS-DOS.
IMS D5205 – Sun 3 systems running SunOS
IMS D4205 – Sun 4 systems running SunOS
IMS D6205 – VAX systems running VMS

How to use the manual

About the manual

The *occam 2 user manual* is divided into two parts, as follows:

- *User Guide and tools 72 TDS 275 02*

    - Chapters 1 to 11 show how the tools are used to develop programs on single transputers and transputer networks.

    - Chapters 12 to 26 provide details of individual tools in terms of command line syntax, command options, running the tool and possible error messages.

- *occam libraries and appendices 72 TDS 276 02*

    - A detailed description is given of all the libraries supplied with the toolset.

    - A number of appendices provide reference material for programmers such as predefined names and constants, transputer instructions, and the implementation of occam on the transputer. A glossary of terms and a short bibliography is also included.

References which span the two parts, take the form of a part number followed by a chapter or section number. Each part contains its own index.

This manual does not contain details of how to install the software, which is to be found in the Delivery Manual that accompanies the shipment.

The manual is intended to cover all host versions of the toolset; where there are differences between the various host implementations, they are highlighted and explained.

Readership

This manual is intended for programmers and system designers who wish to develop transputer programs on host systems. Readers of the manual should already be familiar with programming in a high level language, the software development process, and the general ideas of occam and parallel processing. Familiarity with the syntax of occam will also be an advantage, because occam programs and code fragments are used throughout the book to illustrate concepts and procedures. For information about the occam language, refer to the *'occam 2 Reference Manual'*, which accompanies this release. For an introduction to occam programming, read *'A tutorial introduction to occam*

programming'.

The reader should also be familiar with the hardware and operation of the trans-
puter evaluation board on which the programs will be developed. Information
about INMOS transputer evaluation boards is available in the form of product
datasheets.

User guide

The User Guide, provided in part 1 of this manual, contains information to show
programmers how to use the tools to develop transputer programs. It describes
how to design and build programs for transputers and transputer networks.

Example programs supplied with the toolset are used extensively throughout the
User Guide to illustrate program design and development.

Chapter breakdown

For those who do not wish to read the entire Guide or wish to get started quickly,
some recommendations follow.

If you have not used the toolset before then you should first read chapter 2,
which contains an overview of the toolset.

Chapter 3, 'Getting started' is provided as a tutorial to show users how to compile,
link and run simple occam programs on a single transputer. The example used
is provided in the `examples` directory supplied with the toolset.

Before attempting to write any programs of your own you should read chapters
4 and 8, which show how to compile simple programs that use host terminal i/o.
If you are new to occam you should begin by writing a program which runs on
a single processor before attempting to write multiprocessor code.

Chapter 7 explains how to debug programs running on transputer boards, and
describes how to use the T425 simulator to test programs before loading them
onto hardware. Reading this chapter thoroughly and working studiously through
the examples will help to familiarise you with the operation of the debugger and
simulator tools.

Chapter 9 gives details of how to develop mixed language programs. It shows
how modules written in C can be inserted into an occam program using a set
of library procedures to initialise static and heap areas. Read and digest the
information in this chapter carefully before attempting to write mixed language
programs.

Chapters 10 and 11 provide more specialised information covering the use of

the low level programming and EPROM programming facilities provided with the toolset. These facilities are not aimed at the users who are new to occam or transputers. Users intending to use the EPROM tools should be familiar with INMOS transputers and with memory products.

Tools

The Tools section, provided in part 1 of the manual, contains reference information for all tools in the toolset. Each tool is described in a separate chapter.

The Tools section is not intended to be read in chapter order. Chapters should be consulted as required to obtain information about how to use specific tools.

The occam libraries

Reference information for the occam libraries is given in part 2 of this manual. All the occam library routines provided with the toolset are described. Routines are grouped according to the library to which they belong.

Appendices

These appear at the end of part 2 of the manual. They provide reference information on the following topics:

- Predefined names.

- Transputer instructions.

- Constants.

- The implementation of occam on the transputer.

- Configuration language definition.

- Bootstrap loaders.

- **ITERM**

- Host file server protocol.

A glossary of terms and a short bibliography is also included.

Conventions used in the manual

Convention	Description
Italics	Used in command line syntax to denote parameters for which values *must* be supplied. Also used for book titles and for emphasis.
Bold	Used for new terms, pin signals, and the text of error messages.
`Teletype`	Used for listings of program examples and to denote user input and terminal output.
KEY	Used to denote function keys for the debugger and simulator tools. Keyboard layouts for specific terminals can be found in the Delivery Manual that accompanies the shipment.
▢	Used to indicate the continuation of a function key description.
Braces { }	Used to denote lists of items in command line syntax.
Brackets [ ]	Used to denote optional items in command line syntax.
Option prefix	Examples of command line input are duplicated to show both option prefix characters. Use the line containing the '/' character if you have an MS-DOS or VMS based system and the line containing the '−' character if you are using any other host including UNIX.

User guide

1 Introduction

This chapter gives a gentle introduction to transputers and how transputers are programmed. It introduces the occam model for programming single and multiple transputers, and briefly describes some of its advantages. The chapter also outlines the development process for building and debugging programs, and explains how the tools form an integrated development environment.

1.1   Overview

The occam 2 toolset is a software development system for building and debugging programs on networks of transputers. The occam 2 toolset supports the full range of INMOS transputers and mixed networks of transputers. Used with the ANSI C compiler the occam 2 toolset can be used to build and debug mixed language software systems.

System performance is substantially increased by parallel processing. Transputers and the occam 2 toolset make building high performance parallel systems as simple as sequential programming with conventional microprocessors.

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. It has a very fast (single cycle) on-chip memory, on-chip inter-processor links, and a programmable memory interface that allows external memory to be added with the minimum of supporting logic.

Figure 1.1 shows the architecture of the transputer.

Multi-transputer 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 IMS M212 and T400 each have two high speed communication links). The circuitry to drive these links is all on the transputer chip, and only two wires are needed to connect two transputers together. Figure 1.2 shows four transputers connected using their communication links, and the communication paths between them.

In addition to providing a communication link between programs running on pro-
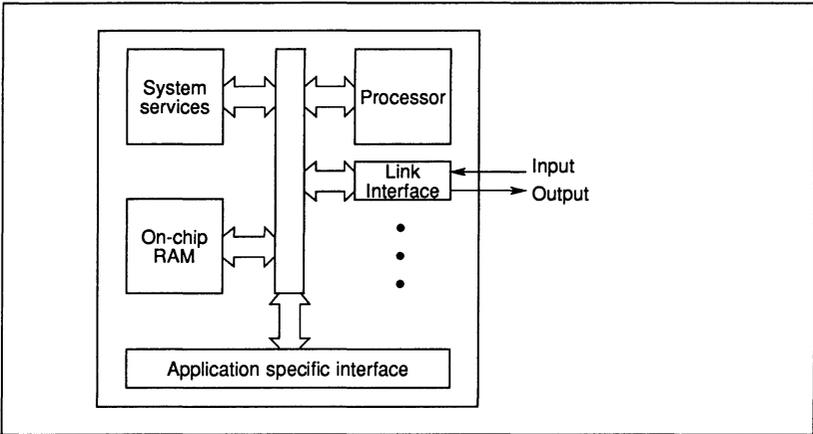
Figure 1.1 Transputer architecture

cessors, transputer links allow memory to be examined without loading a program, and permit programs to be loaded and executed. This allows whole networks of transputers to be loaded down a single transputer link.
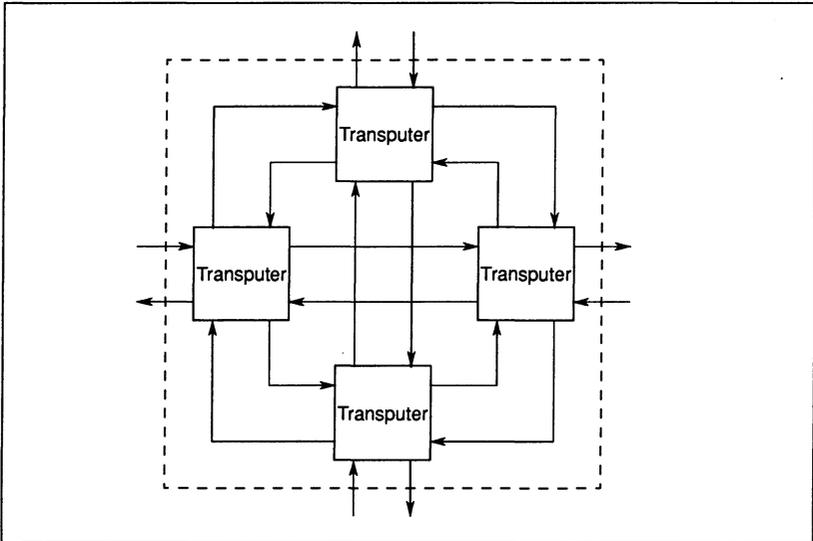
Figure 1.2 A node of four transputers

Each single transputer supports parallel processing through a system of internal

channels implemented as words in memory. Each transputer has a highly efficient built-in run-time scheduler; processes waiting for input or output, or waiting on a timer consume no CPU resources, and process context switching time on an IMS T800-25 is less than one microsecond. The communication links operate concurrently with the processing unit and can transfer data on all links without affecting the performance of the CPU.

The range of transputer devices available includes: 32 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 suppliers for a variety of hosts. These boards can be used for:

- Developing and debugging transputer software

- Running transputer programs (as accelerator boards)

- Loading software to transputer networks from the host.

- Building specific transputer networks.

1.3 Transputers and occam

occam 2 has been designed to reflect the architecture of the transputer, and for maximum coding efficiency the whole system can be programmed in occam 2. The inherent security and code efficiency of occam and the ability to use the special features of the transputer make occam 2 a powerful tool for programming concurrent systems.

Transputers can also be programmed in C and FORTRAN, and their optimised design ensures efficient code. Where programs need to exploit concurrency but still need to use languages other than occam 2, special occam code can be used to link modules together.

1.3.1 The occam programming model

The occam programming model consists of parallel processes communicating through channels. Channels connect pairs of processes and allow data to be exchanged between them. Each process can be built from a number of 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 synchronized. When a message is passed

between two processes the output process does not proceed until the input process is ready. Buffered communication can be achieved by explicitly inserting a buffer process between the two processes.

The occam programming model also provides an excellent basis for building mixed language systems. Components written in languages other than occam can be defined as processes inputting and outputting messages on channels. The ANSI C and FORTRAN compilers supplied by INMOS are compatible with occam and can be used to build equivalent occam processes in any of these languages. Library functions are provided in each language for the input and output of messages on channels.

1.3.2　Multitransputer programming

In the occam 2 programming language parallelism can be expressed directly. Each occam process is an independently executable process. A configuration language extension to occam 2 is used to distribute processes over networks of transputers, and can be used to program multi-processor systems.

Figure 1.3 shows how three discrete processes, programmed in occam or in a compatible language, can be executed on a single processor or on three processors connected in series.
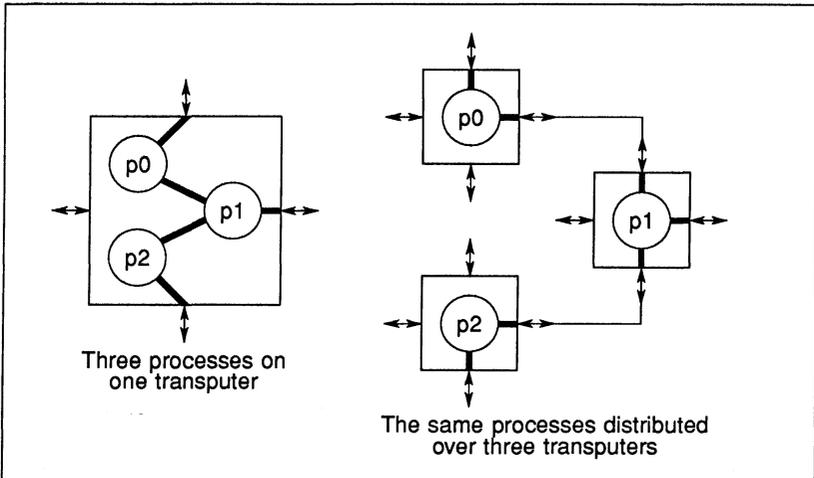
Figure 1.3 Mapping processes onto one or several transputers

1.3.3 Reliability

Because it has a formal mathematical framework, the occam 2 language can be extensively checked at compile time, and many programming errors can be detected before the program is run. This significantly improves the reliability of programs, and makes building correct programs faster and easier.

Each construct in the language has a precise meaning. This makes programs easier to write and understand, and supports the formal mathematical manipulation of programs required for program proving and advanced program optimisation techniques.

1.3.4 Real time programming

occam 2 provides specific support for real time programming. The key features of the transputer that support real time programming are listed below.

- Direct and efficient implementation of parallel processes in hardware

- Prioritisation of parallel processes

- Implementation of software interrupts as messages on occam channels, so that interrupt routines can be written as high priority processes

- Easy programming of software timers, allowing delays and non-busy polling

1.4 Program development using the toolset

The occam 2 toolset is a complete set of cross-development tools. The tools run under standard host operating systems, either on the host itself or on a transputer attached to the host, and use standard ASCII source files. All the tools can be used in conjunction with existing software for text editing and source control and with compilation utilities such as Make programs. For embedded systems, programs can be loaded onto the target hardware from the host via a transputer link.

1.4.1 System design

The designer can use the occam programming model to design software systems at the application level, by identifying the separate components of the system in terms of processes and collections of related functions and procedures. The design can be directly expressed in occam and then checked by the com-

piler before transferring it to hardware.

1.4.2   Programming and code generation

To implement components of the design the programmer creates occam source texts, then compiles and links them together to produce executable code. Compiled source files can easily be combined into libraries for code sharing.

Code is linked using the toolset linker. For multi-transputer systems software processes are allocated to transputers, and channels are allocated to links, in a configuration description. This description, plus the linked code for each transputer, is processed by the toolset configurer to create a multi-transputer program. This program can then be distributed across a transputer network down transputer links.

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.

occam programs can be executed and tested without transputer hardware using the T425 simulator tool which provides low-level debugging facilities. This method is appropriate for debugging individual parts of a large transputer program that would run on a single T425 processor.

2 Overview of the toolset

This chapter introduces the toolset and briefly describes each of the tools in turn. It also introduces the occam libraries, describes host system dependencies, and explains the conventions used within the toolset.

2.1 Introduction

The occam 2 toolset is a set of tools and supporting software that help with the development of transputer programs. It allows programs developed on host machines to be loaded onto transputers and transputer networks via transputer evaluation boards such as the IMS B004 and B008 boards. All of the tools operate with files in standard host format. This enables you to use the editor with which you are familiar, and allows different types of version control systems to be used.

A list of the tools in the toolset is given in table 2.1.

There are a number of different implementations of the toolset, running on different host computers. Versions are available for the IBM PC/AT and PC/XT (and compatibles) running DOS, DEC VAX systems running VMS, and the Sun Microsystems Sun 3/Sun 4 workstations running SunOS.

This manual covers all host versions of the toolset. Where differences exist between implementations they are highlighted and explained.

2.1.1 Standard file format

All tools in the occam 2 toolset generate and use files in a standard object code format known as TCOFF (*Transputer Common Object File Format*). The adoption of this format makes the mixing of code from different compilers more convenient and facilitates the porting and transfer of object code. In particular, it enables code generated by all INMOS language compilers to be mixed in the same system.

Support is provided for previous versions of the toolset (i.e. the IMS D705, D605 and D505 products) which used a file format known as LFF *Linker File Format*. It is recommended, however, that the current release of the toolset is used to recompile existing programs. This has the advantage that the current toolset may be used for their further development.

Two levels of support are provided for LFF format:

- The linker tool `ilink` supplied with the current toolset supports the pro-
duction of files in LFF format. The tool has a command line option which
enables output files to be produced that can be used with the `iboot`
and `iconf` tools issued with the IMS D705, D605 and D505 releases of
the toolset.

- A file convertor tool `icvlink` supplied with the current toolset enables
code generated by previous INMOS toolsets (i.e. the IMS D705, D605,
D505, D511A, D611A and D711A products) to be used with the current
occam 2 toolset. **Note**: there are some limitations on this tool's use,
see chapter 13.

2.1.2   New configuration language

The toolset introduces a modified configuration language that allows software
and hardware to be described separately and joined by a mapping description.
The language is an extension to occam and can be used on any size of network.

Program	Description
icollect	The code collector. Takes the output of the configurer tool and generates bootable code for a transputer network.
icvemit	Memory interface file convertor. The tool converts files produced by iemi (a previous version of iemit) into the file format recognised by the current version of ieprom and iemit.
icvlink	The TCOFF file format convertor. Converts LFF object files to TCOFF format.
idebug	The toolset debugger. Provides symbolic and assembly level debugging.
idump	The memory dumper for storing the contents of the root transputer. Used when debugging programs running on the root transputer.
iemit	The transputer memory configuration tool. Used for evaluating and defining specific memory configurations for incorporation into ROM programs.
ieprom	The EPROM program formatter tool. Formats transputer bootable code for ROM programmers.
ilibr	The librarian. Builds libraries of compiled code.
ilink	The linker. Resolves external references and links compiled code into a single file.
ilist	The binary lister. Displays source level information from object code.
imakef	The Makefile generator. Generates Makefiles for building object and bootable code. Also creates library usage files.
iserver	The host file server. Loads programs onto transputer boards and provides run-time communications with the host.
isim	The T425 transputer simulator.
iskip	The skip loader tool. Prepares transputer networks to run programs without using the root transputer.
oc	The occam compiler. Compiles source for IMS T212, M212, T222, T225, T400, T414, T425, T800, T801 and T805 transputers.
occonf	The configurer. Checks the configuration description and produces a data file for the code collector.

Table 2.1 The occam 2 toolset

2.2   oc – the occam 2 compiler

The occam 2 compiler takes as input occam source code contained within
standard host format text files. Any text editor that produces standard ASCII
files can be used to create the occam source.

The compiler produces code for T212, M212, T222, T225, T400, T414, T425,
T800, T801 or the T805 transputers in one of three program execution error
modes. Command line options allow you to specify the transputer type, error
mode, and other information required by the compiler.

The compiler supports a number of source code directives which enable different
types of source files to be compiled together. The main directives are:

- #INCLUDE - includes other source files

- #USE - uses separately compiled code and libraries

The compiler also supports the directives #COMMENT, #IMPORT, #OPTION,
and #PRAGMA.

The implementation of transputer classes, error modes and channels have all
changed slightly with this release of the toolset. Details of these changes are
given in chapters 4 and 25.

2.3   Code generation tools

Three tools are used in sequence to generate the executable file for a transputer
or transputer network from the compiled object code:

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

occonf – the configurer which generates a configuration data file for
multitransputer programs. This tool is not needed for single transputer
programs.

icollect – the code collector which reads the configuration file and
generates a single bootable file for a transputer network. The collector is
also used to create a bootable file from linker output for a single transputer
program.

2.3.1 Linker

The linker `ilink` links separately compiled modules and libraries into a single file, resolving external references and generating a *linked unit*.

Linked units can be used in configuration descriptions to map software onto specific arrangements of transputers. While a linked unit for a single transputer program can used by `icollect` to generate a bootable file.

2.3.2 Configurer

The configurer `occonf` generates configuration information for transputer networks from a *configuration description*, written using 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 collector tool.

2.3.3 Collector

The code collector tool `icollect` takes the data file generated by `occonf` and generates a single file that can be loaded on a transputer network. The file contains bootable code modules for each processor on the network, along with distribution information that is used by the loader.

`icollect` is also used to generate bootable code for single transputer programs from linked units. This mode of use must be selected by a command line option.

2.4 Code loading

Bootable code for single transputers and transputer networks is loaded onto the transputer hardware using the host file server tool `iserver`. The auxiliary skip loading tool `iskip` can be used prior to `iserver` in order to load a program onto an external network connected via a root transputer.

2.4.1 Host file server

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. `iserver` runs on the host computer, which is usually not a transputer.

2.4.2   Skip loader

The skip loader `iskip` is used to force a program to be loaded over the root transputer (the transputer connected to the host). This is useful for loading programs onto a transputer board connected to the host via a root transputer.

It is also useful for debugging programs that normally use the root transputer to run all or part of a program. The debugger always runs on the root transputer. Provided the network has at least one processor which is not used by the program, `iskip` may be used in conjunction with `iserver` to load the program over the root transputer. For details of skip loading see section 6.6.

2.5   Program development and support tools

Seven tools are provided to assist and support program development:

`idebug` – the 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.

`icvlink` – the file format convertor which allows import of code from earlier INMOS toolsets.

`imakef` – the Makefile generator which creates Makefiles for use with MAKE programs.

`isim` – the T425 simulator tool which enables programs to be executed in the absence of transputer hardware.

2.5.1   Network debugger

The network debugger `idebug` provides comprehensive debugging facilities 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 debugging with breakpoints. In breakpoint mode variables can be inspected and modified and debugging messages can be inserted in any process on the network.

2.5.2   Memory dumper

The special debugging tool `idump` is provided to assist with the post-mortem debugging of programs that run on the root transputer. When used in post-mortem mode `idebug` executes on the root transputer and overwrites the program image. `idump` saves the program image to a file which is later read by the debugger.

2.5.3   Librarian

The librarian `ilibr` creates libraries of compiled code that can be used in application programs. Each separately compiled file, that is supplied as input to the librarian, becomes a library module that can be selectively linked.

A library can contain modules compiled from the same source for different targets and compilation modes.

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

2.5.4   Binary lister

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 library contents, code entry points, and external reference data.

2.5.5   Makefile generator

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 object file and generates a Makefile in standard format. In order to make use of the tool a special set of file extensions for source and object files must be used throughout program development.

2.5.6   File format convertor

The file format convertor `icvlink` converts object files generated by earlier INMOS toolsets to TCOFF format. TCOFF is a standardised object file format for transputer code.

`icvlink` allows existing object code be used with the current toolset. Files to be converted must be *compiled* or *linked* object files or libraries.

2.5.7   T425 simulator

The T425 simulator tool `isim` simulates the operation of the T425 transputer, enabling programs to be executed in the absence of transputer hardware. It provides low-level debugging features such as the inspection of variables, registers, and queues, disassembly of memory, break points, and single step execution.

2.6   EPROM support tools

Three tools provide support for installing bootable transputer code in ROM: the EPROM programmer `ieprom`; the memory configurer `iemit` and `icvemit` the memory interface file convertor.

2.6.1   EPROM programmer

The EPROM programmer `ieprom` converts bootable files into a format suitable for input to ROM programmers. Files are generated for input to several proprietary ROM loaders, or in hexadecimal or binary format.

2.6.2   Memory configurer

The memory configurer `iemit` allows specific memory configurations to be evaluated and tested 'on the bench' before committing them to a device. The completed configuration can be included in the `ieprom` output file for automatic installation into the processor.

2.6.3   Memory interface file convertor

`icvemit` is an auxiliary tool that converts files produced by `iemi` (a previous version of `iemit`) into the file format recognised by the current version of `ieprom` and `iemit`. See section 16.8 for further details.

2.7 The occam libraries

A comprehensive set of libraries and include files are provided with the toolset. Some form part of the standard support for the occam language (the compiler libraries), others are user-level libraries to support standard programming tasks such as terminal i/o and file access.

Object code is supplied for all libraries and in some cases source code is supplied as well. Table 2.2 lists the libraries that are supplied with the toolset and specifies whether the source code is provided. Details of all the libraries can be found in part 2, chapter 1.

Library	Description	Source provided
occam*x*.lib	Compiler libraries	no
hostio.lib	General purpose i/o library	yes
streamio.lib	Stream i/o support	yes
snglmath.lib	Single length maths functions	yes
dblmath.lib	Double length maths functions	yes
tbmaths.lib	T400/T414/T425 optimised maths functions	yes
string.lib	String handling routines	yes
xlink.lib	Extraordinary link handling routines	no
convert.lib	Type conversion routines	yes
crc.lib	CRC coding	no
debug.lib	Debugging support	no
callc.lib	Mixed languages support	no
msdos.lib	DOS specific hostio library	yes

Table 2.2 The occam 2 libraries

2.7.1 Constants

Files containing definitions of constants and protocols are also provided for use with the occam libraries. These are listed in table 2.3.

2.7.2 Compiler libraries

The compiler libraries are used internally by code generated by the compiler; they are not intended for direct use by the programmer.

File	Description
`hostio.inc`	Host file server constants
`streamio.inc`	Stream i/o constants
`mathvals.inc`	Mathematical constants
`linkaddr.inc`	Transputer link addresses
`ticks.inc`	Rates of the two transputer clocks
`msdos.inc`	DOS specific constants

Table 2.3 Library constants

The compiler automatically loads the library required for a specific combination of compiler options.

2.7.3   Maths libraries

The maths libraries provide trigonometric and logarithmic functions for all transputer types supported by the toolset. Single and double length routines are supplied in the libraries `snglmath.lib` and `dblmath.lib` respectively, and versions of the same routines optimised for the T400, T414 and T425 processors are provided in the library `tbmaths.lib`. Constants for the maths libraries can be found in the include file `mathvals.inc`.

2.7.4   I/O libraries

Two libraries containing routines to assist with i/o are provided with the toolset. Constants for the two libraries are provided in separate files.

Hostio library

The hostio library contains routines that provide access to the file system and other host services via the host file server. The routines are based on communication with the server via the `SP` protocol. The `SP` protocol is defined in the include file `hostio.inc`.

The hostio library is used for:

- File handling

- Host access

- Terminal i/o

Other routines provide facilities such as time and date processing, process buffer-

ing and multiplexing.

Streamio library

The streamio library contains routines which provide i/o at a higher level than the hostio routines. The protocol is based on a stream model. The streamio library is used for general character-based i/o using stream protocols, and for controlling the screen display. Protocols for the streamio library are defined in the include file **streamio.inc**.

Stream input and output procedures are used to input and output characters using keystream and screen stream protocols. These protocols must be converted to the server protocol before communicating with the host. For this reason the streamio routines include stream processes to perform this conversion.

2.7.5   Other libraries

String handling library

The string handling library provides string handling functions and procedures to perform, for example, string comparison, string search, string editing, and line parsing.

Type conversion library

The type conversion library converts occam data types to ASCII strings and vice versa.

Extraordinary link handling library

The extraordinary link handling library provides facilities for handling error situations on links.

Block CRC library

The block CRC library provides functions for generating CRC codes from character strings.

Debugging support library

The debugging support library provides functions for stopping processes, inserting debugging messages and analysing deadlocks.

Mixed language support library

The mixed languages support library provides functions for initialising static and heap areas, enabling modules written in sequential languages to be incorporated in occam programs.

DOS specific hostio library

The DOS specific hostio library supports the use of functions specific to the IBM PC and other DOS hosts.

2.8   Program development

The occam 2 toolset is a development system for transputers. Creation of transputer executable code involves several stages which involve the use of specific tools for each stage of the process.

The main steps in developing a program, for a transputer or transputer network, and the tools to use at each stage are listed below.

- **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. occam source code must conform to the occam 2 language definition.

- **Compile the source:** Each occam source code file must be compiled using the occam 2 compiler oc to produce one or more compiled object files in TCOFF format. Each file must be compiled for the same or a compatible transputer type and compilation mode.

- **Link the compiled units:** Compiled source files are linked together. This generates a single file called a *linked unit* with no external references. The linking operation also links in the library modules required by the program, which are selected by transputer type and compilation mode from the compiled library code.

- **Configure the program:** For multitransputer programs a configuration description must be constructed 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 occonf to produce a configuration data file.

- **Generate executable code:** The configuration data file generated by occonf is analysed by the code collector icollect to generate a single executable file for a transputer network. The same tool is used

to create bootable programs for running on a single transputer, by using the linked unit as the input file.

- **Load and run the program:** The executable or *bootable* file is loaded onto the transputer network down a host link. Once loaded into memory, the code begins to execute.

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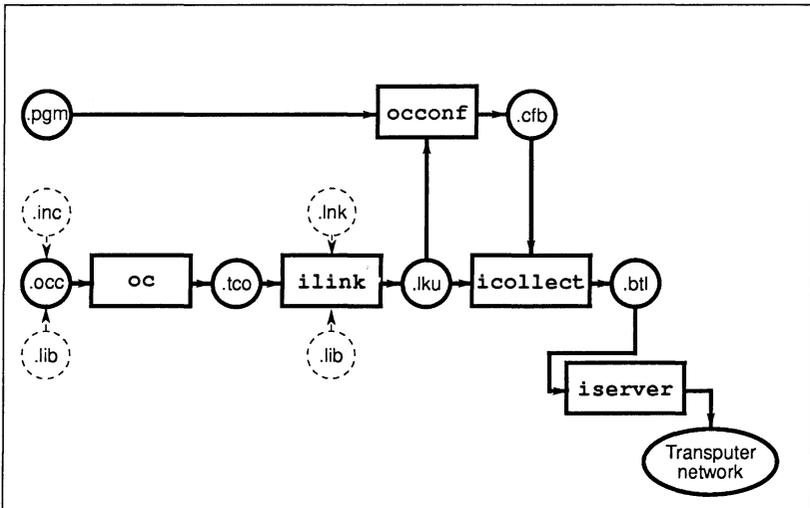
Figure 2.1 Toolset compilation architecture

2.8.1   Development support

The development support tools are aids to developing software and software systems. They are designed to assist with processes such as debugging, code sharing, and software version control. Used systematically during software development they can help to produce reliable code quickly and with the minimum of manual recompilation.

The network debugger `idebug` can be used in breakpoint mode during code development to test and debug programs interactively. In post-mortem mode it can be used to investigate the reasons for program failure.

The librarian `ilibr` can be used to build libraries which make it possible to share and transfer code between developers.

The binary lister `ilist` can be used throughout program development to assess code size and structure, and to determine the contents of object code files such as libraries.

The Makefile generator `imakef` can be used in conjunction with a MAKE program to ensure that all object code is updated to reflect changes in source files.

The file format convertor `icvlink` can be used to import existing object code where the source is unavailable or too complicated to recompile easily.

The T425 simulator `isim` can be used to run programs in the absence of transputer hardware. It provides low-level debugging features such as the inspection of variables, registers, and queues, disassembly of memory, break points, and single step execution.

2.9   File extensions

File extensions can be used to indicate the various types of source and object code that they contain and certain default names are assumed and where possible generated by the tools. For example, the compiler assumes the suffix `.occ` for the input source file and adds the extension `.tco` to the output file unless otherwise specified.

Assumed extensions permit common input file extensions to be omitted on the command line and default generated extensions allow output files to be easily identified and manipulated by the host file system.

The default output extensions and assumed input extensions are not part of the required syntax and may be modified or omitted by personal choice (except when `imakef` is used, see below). None are mandatory parts of the syntax.

Adoption of a convention is recommended where large systems are being developed. The standard set of conventions outlined here can be used, or a separate system can be designed to suit a particular environment. The standard toolset system has the advantage of built in defaults, and has been designed to reflect the underlying architecture of the toolset.

The default extensions are listed in table 2.4 and the relationships of the extensions to the compilation architecture is illustrated in figure 2.1.

File extensions for use with `imakef`

The Makefile generator `imakef` requires special file extensions for compiled and linked object files, which differ from the set of default extensions presented here.

Ext	Description
`.btl`	Bootable code file. Created by `icollect`.
`.btr`	Executable code minus bootstrap information used for input to `ieprom`. Created by `icollect`.
`.cfb`	Configuration data file. Created by `occonf` and `icollect`.
`.clu`	Configuration object file. Created by `occonf`.
`.dmp`	Core-dump file created by `idump` or network-dump file created by `idebug`.
`.inc`	Include file. Input to `oc` and `occonf`.
`.lku`	Linked unit. Created by `ilink`.
`.lbb`	Library build file. Input to `ilibr`.
`.lib`	Library file. Created by `ilibr`.
`.liu`	Library usage file. Created and used by `imakef`.
`.lnk`	Linker indirect file. Input to `ilink`.
`.occ`	occam 2 source files. Assumed by `oc`.
`.pgm`	Configuration description. Assumed by `occonf`.
`.rsc`	Dynamically loadable code file. These files are designed to be executed by `KERNEL.RUN`.
`.tco`	Compiled code file. Created by `oc`.

Table 2.4 Standard file extensions

The extensions are used by `imakef` to trace file dependencies and construct the necessary commands for building all types of object files.

If you use `imakef` then you *must* use the special set of extensions. For more details see chapter 21.

2.10 Host dependencies

The occam 2 toolset is hosted on four systems:

- IBM PC running DOS

- DEC VAX running VMS

- Sun 3 running SunOS (UNIX)

• Sun 4 running SunOS (UNIX)

Source and object code is portable between all these systems.

The four implementations have been designed to reflect the 'flavour' of the operating system. This leads to minor differences between them in the areas of command line syntax and the filename character set. Installation issues such as the setting of environment variables and the definition of search paths are also host dependent, and are covered in detail in the Delivery Manual that accompanies the release. They are only described briefly here.

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

Command line syntax

The major difference between different host implementations is the use of the standard host system option prefix. For MS-DOS and VMS based toolsets the prefix character is the forward slash '/'. For all other hosts, including UNIX the prefix character is the dash '−';

2.10.1   Libraries

Most library routines supplied with the toolset are host independent, but a few specific procedures may be provided for some operating systems. For details of host dependent routines see the Delivery Manual.

If you wish to write programs that will be fully portable across different systems, use only the host independent routines, which are described in part 2, chapter 1.

2.10.2   Filenames

Filenames, with or without the full directory path, conform to the normal conventions of the host operating system except that characters which can be interpreted as directory separators must not be used in the filename part. Prohibited characters are: dot ., colon :, semi-colon ;, square brackets [ ], round brackets ( ), angle brackets <>, forward slash /, backslash \, exclamation mark !, or the equals sign =.

Where the host operating system allows logical names to be used in place of filenames, such as with VMS, the toolset allows logical names to be used, but the name must be followed by a dot (.). This prevents the tool from adding an extension, which would generate a host file system error.

2.10.3 Search paths

All tools which use or generate filenames use a standard mechanism for locating
files on the host system. This mechanism is used whenever a filename has to be
interpreted e.g. from the command line, as part of a directive such as #INCLUDE
or #USE or a library call. The same mechanism is used in all operating system
versions of the toolset.

The mechanism is based on a list of directories to be searched. If the name
includes a directory path only this directory is searched. If the file is not found
on the path an error is generated. Relative pathnames are treated as relative to
the current directory i.e. the directory from which the tool is invoked.

If no directory path is specified the current directory is searched followed by the
directories specified in the ISEARCH environment variable e.g.

ISEARCH=C:\IOCTOOLS\LIBS\;C:\MYDIR

The mechanism used to define environment variables depends 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 showing how to set up environment variables on your system can be
found in the Delivery Manual that accompanies the release. Details of the oper-
ating system commands can be found in the operating system documentation.

2.10.4 Host environment variables

The toolset uses several 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 list of directories that will be searched if the full pathname is not specified. Pathnames must be terminated by the directory separator character. Used by all tools that read and write files.
`ISERVER`	Defines an alternative iserver to be used by the host system for booting a transputer network and communicating with the application program running on the network.
`ITERM`	The file that defines terminal keyboard and screen control codes. Used by `idebug`, `isim` and `iemit`.
`IBOARDSIZE`	The size (in bytes) of memory on the transputer board. Used by non-configured programs.
`TRANSPUTER`	The host address at which the transputer board is connected to the host. Used by `iserver`.
`IDEBUGSIZE`	The amount of memory (in bytes) on the root transputer available for use by `idebug`.

2.10.5   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 `ilist` is `ILISTARG`.

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

Tool	Variable
`ilink`	`ILINKARG`
`ilibr`	`ILIBRARG`
`ilist`	`ILISTARG`
`icvlink`	`ICVLINKARG`

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

2.11   Toolset conventions

The toolset conforms to a number of conventions for the command line syntax, file names, and error reporting.

2.12   Command line syntax

All tools in the toolset conform to a standard command line syntax. Toolset commands use the following syntax 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 MS-DOS and VMS based toolsets and ('–' for all other hosts including UNIX).

- 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.

Standard options

Where options are common to more than one tool in the toolset, the following conventions apply:

- All tools provide help information if invoked with no options.

- The 'F' option, where supported, specifies an indirect input file. If no name is given then input may be taken either from host standard input (normally the keyboard) or the command line.

- The 'I' option, where supported, displays progress information as the tool runs.

- The 'L' option , where supported, loads the tool onto a transputer board and awaits a command line. Only applies to transputer hosted tools.

- The 'O' option, where supported, is used to specify an output filename. If no filename is given then ASCII output is sent to host standard output (normally the screen), or to a file whose name is derived from an input file.

- The 'XO' option, 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. Single invocation is the default.

- The 'XM' option, 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. When the tool is reinvoked, a combination of server options and the tool's own options are used on the server's command line. For example:

UNIX system:

```
oc -l -xm
iserver -ss -xm myprog.occ -o myprog.t4h
iserver -ss -xm -t8 myprog.occ -o myprog.t8h
iserver -ss -xm -t5 myprog.occ -o myprog.t5h
```

MS-DOS/VMS system:

```
oc /l /xm
iserver /ss /xm myprog.occ /o myprog.t4h
iserver /ss /xm /t8 myprog.occ /o myprog.t8h
iserver /ss /xm /t5 myprog.occ /o myprog.t5h
```

In this example the occam compiler is first loaded onto the transputer board. It is then invoked in continuous execution mode, with different compiler options (see section 25) selected for the program "myprog.occ". A different output file is specified each time the tool is invoked. The server 'ss' option enables the program to communicate with the host file server (see chapter 22). The 'xm' option must be used each time the tool is to be reinvoked.

2.12.1 Error handling and message format

All tools in the toolset use a common system of error handling and a common format for error messages. This has the following advantages:

- The tool generating the error can be identified even when the tool is run in a 'background' mode, that is, out of contact with the terminal.

- Some editors can provide automatic location of the error if the error messages are in a fixed format.

- Host programs or operating system utilities can be used to detect errors.

The format includes information to assist in locating the error in the file, an indication of the error severity, and a message explaining why the error occurred. The general format is as follows:

> *severity–toolname –filename (linenumber)–message*

where: *severity* indicates the category of error, which can be: *Warning*; *Error*; *Serious*; or *Fatal*. These are described in more detail 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 was detected. They are only displayed when 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 is the text explaining why the error occurred and, if appropriate, how to recover from the problem.

For example:

Error–oc–Invalid command line option (*sting*)

Severities

Warning messages identify relatively minor inconsistencies in code; they may also warn of the impending generation of more serious errors. The tool continues to run and may produce usable output if no errors of a more serious nature are encountered subsequently.

Error messages indicate errors from which immediate recovery is possible but long term recovery is unlikely. The tool may continue to run, but further errors are likely and the tool will probably abort eventually. No output is produced.

Serious messages indicate errors from which no immediate recovery is possible. Further processing is abandoned and the operation is aborted. No output is produced.

Fatal errors indicate internal inconsistencies in the software and cause immediate termination. No output is produced. Fatal errors should be reported immediately to an INMOS field applications engineer.

Information messages

Messages that are part of the normal operation of the tool, for example, infor-
mation from the debugger and simulator tools are displayed in special formats.
The formats will become familiar with use.

3 Getting started

This chapter contains a tutorial that shows you how to compile, link, and run a simple example program on a single processor.

A more complex programming example, illustrating separate compilation, can be found in chapter 4, together with a detailed description of program development for single transputers. While chapter 5 provides a description and examples of multitransputer programming.

The tutorial, given in this chapter, assumes that you have a boot from link board containing a IMS T400, T414 or T425 processor. If you have a board fitted with any other transputer you must compile and link the program for that transputer type, see section 3.3.6. The tutorial also assumes that certain environment variables have been set up. These are introduced in sections 2.10.3 and 2.10.4 and a description of how to set them up is given in the delivery manual supplied with this product.

If you do not have a transputer board use the T425 simulator tool **isim** to run the application program, see section 3.3.5.

3.1 Example command line

Where necessary, the 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 Interrupting programs

To interrupt an application program while it is still running, press the host system BREAK key to interrupt the server. See the delivery manual, section 'Server Interrupts' for further details.

When the BREAK key is pressed the following prompt is displayed:

(x)exit, (s)hell, or (c)ontinue?

To abort the program type 'x' or press RETURN This terminates the host file server.

To suspend the program so that you can resume it later, type 's'.

To abort the interrupt and continue running the program, type 'c'.

3.3   Compiling and running a simple example program

The example program `simple.occ` reads a name from the keyboard and displays a greeting on the screen. The source of the program can be found in the toolset `examples` directory. The program uses the library `hostio.lib` and incorporates the include file `hostio.inc`.

The program is illustrated below.

```
#INCLUDE "hostio.inc" -- contains SP protocol

PROC simple (CHAN OF SP fs, ts, []INT memory)

  #USE "hostio.lib"  -- iserver libraries

  []BYTE buffer  RETYPES memory:

  BYTE result:
  INT length:
  SEQ
    so.write.string      (fs, ts,
                          "Please type your name :")
    so.read.echo.line    (fs, ts, length, buffer,
                          result)
    so.write.nl          (fs, ts)
    so.write.string      (fs, ts, "Hello ")
    so.write.string.nl   (fs, ts,
                          [buffer FROM 0 FOR length])
    so.exit              (fs, ts, sps.success)
:
```

The first line in the program loads the file `hostio.inc`. This file contains the definition of protocol `SP`, used to communicate with the host file server, and a number of constants that are used in conjunction with the host i/o library.

The procedure `simple` is then declared. All the working code is contained within this procedure. Single processor programs must always use a similar parameter list.

The server library `hostio.lib` is referenced by the #USE directive. This library contains all the procedures used by the program. See part 2, section 1.4 for descriptions of the routines.

Before the body of the procedure a number of variables are declared. First, the **memory** array is retyped as a **BYTE** array. This enables the program to use the free memory on the board as a character buffer.

The variables **length** and **result** are then declared for use by the program. The variable **length** refers to the number of characters in the name read from the keyboard, and **result** is used by the library routine to indicate whether or not the read was successful. The result is ignored by this example for the sake of simplicity; it is assumed that screen writes and keyboard reads always succeed.

The working code is contained within a **SEQ**, indicating that the statements which follow are to be executed sequentially. All of the statements are calls to library routines in **hostio.lib**. The code prompts for a name, reads the name from the keyboard, and types a greeting on the screen.

The last statement calls a library procedure which terminates the server, returning control to the host operating system. Without this statement the program would finish and appear to hang, and the server would have to be terminated explicitly by interrupting the program.

3.3.1   Setting environment variables

Certain environment variables must be set up prior to using the toolset. These are introduced in sections 2.10.3 and 2.10.4 and a description of how to set them up is given in the delivery manual supplied with this product. For example, the compilation will fail with a message indicating that **hostio.inc** has not been found, should the environment variable **ISEARCH** not be set up correctly.

3.3.2   Compiling the example program

In order to compile the program in its simplest form i.e. with all defaults enabled the following command line should be used:

```
oc simple
```

Because the file has the default extension of **.occ** you can omit it when invoking the compiler.

The compiler will create a file called **simple.tco**, containing the code compiled for a T414 in HALT mode. The compiler will perform the necessary syntax, alias and usage checks and will insert code to perform run-time error checking. By default the compiler enables interactive debugging with **idebug**.

3.3.3   Linking the example program

To use the result of your compilation it must be linked with the libraries that it uses.

To link the program type:

```
ilink simple.tco hostio.lib -f occama.lnk        (UNIX)
ilink simple.tco hostio.lib /f occama.lnk    (MS-DOS/VMS)
```

The linked program will be written to the file **simple.lku**. As no output file is specified, the file is named after the input file and the default link extension **.lku** is added.

The library **hostio.lib** is the server library used by this program.

The '**f**' option specifies a linker indirect file containing commands and directives to **ilink**. Three indirect files are supplied to support different transputer types. They are **occam2.lnk**, **occama.lnk** and **occam8.lnk**; they are described in chapter 19. These files identify various libraries including compiler libraries which are required to be linked with the program. These files are provided as a short-hand method of specifying such libraries to the linker.

The file **occama.lnk** is the correct file to use for T4 series transputers.

Note: In more complex programs, libraries may be dependent on other files and libraries. To ensure all necessary libraries are linked into a program, the **imakef** tool may be used with a suitable MAKE program. (See below).

3.3.4   Creating a bootable file

Before the program can be run it must be made 'bootable'. This involves adding bootstrap information to make the program loadable and is achieved using the collector tool **icollect**. One of the following commands should be used depending on the type of host in use.

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

By default **icollect** expects the input file to have been produced by the configurer. Because the example program is going to run on a single processor there is no need to configure it. The '**t**' option instructs the collector to build a bootable file from a linked unit. The bootable program will be written to the file **simple.btl**.

`icollect` will also create a configuration binary file as a by-product of creating the bootstrap. Configuration binary files describe the network configuration, in this case a single transputer. This file will have the extension `.cfb` and is created by `icollect` for use by the debugger. For multitransputer programs the configurer is used to create configuration binary files.

Chapter 12 gives more information on the collector tool.

3.3.5 Running the example program

To run the program it must be loaded onto a transputer board using the host file server tool `iserver`. To load and run the program use one of the following commands:

```
iserver -se -sb simple.btl              (UNIX)
iserver /se /sb simple.btl            (MS-DOS/VMS)
```

The '`sb`' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The '`se`' option directs the server to terminate if the program sets the error flag. For more details about the server options see chapter 22.

Figure 3.1 shows an example of the screen display, obtained by running `simple.btl` on a UNIX based toolset, for a user called 'John'.
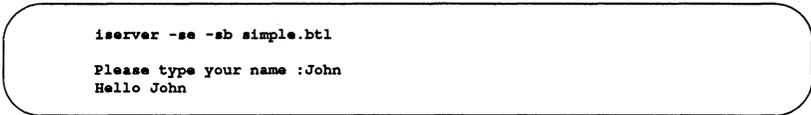
```
iserver -se -sb simple.btl

Please type your name :John
Hello John
```

Figure 3.1 Example output produced by running `simple.btl`.

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.btl                      UNIX
isim /bq simple.btl                    MS-DOS/VMS
```

The '`bq`' option specifies batch quiet mode which causes the simulator to run the program and then terminate. For more details about how to use the simulator see chapter 23.

3.3.6   Compiling and linking for other transputer types

If you are using a transputer other than a T400, T414 or T425 you must specify a
target transputer type for the compilation and linkage function, since the default
type T414 will be inappropriate. Chapters 25 and 19 describe the options avail-
able. The same option must be specified to both the compiler and the linker,
otherwise the linker will report an error. In addition, you must change the linker
indirect file as described in chapter 19.

For example to compile and link the program 'simple.occ' so that it will run
on a T800, T801 or T805 use the following command lines, as appropriate:

UNIX hosts:

```
oc simple -t800
ilink simple.tco hostio.lib -f occam8.lnk -t800
```

MS-DOS/VMS hosts:

```
oc simple /t800
ilink simple.tco hostio.lib /f occam8.lnk /t800
```

3.4   Using `imakef`

As an alternative method of program development the toolset Makefile generator
`imakef` can be used. This tool can produce a Makefile for any type of file that
can be built with the toolset tools. `imakef` serves two purposes:

- It enables the user to generate a target file automatically (e.g. a bootable
  file) without having to manually perform the intermediate stages of pro-
  gram development i.e. compiling, linking, configuring etc.

- For more complex programs, comprising several modules, it simplifies
  the incorporation of changes to the program by identifying dependencies
  and incorporating them into the Makefile.

In order for `imakef` to be able to identify file types, a different system of file
extensions must be used to that used in the examples above. See chapter 21
for a description of `imakef` and the extensions used.

To create a Makefile for the example program, use the following command:

```
imakef simple.b4h
```

The `.b4h` extension informs **imakef** that we wish to build a bootable program for a T414 in the default HALT error mode. **imakef** will create a Makefile called **simple.mak** containing full instructions on how to build the program.

To build the program run the MAKE program on **simple.mak**. The entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

For example:

```
make -f simple.mak                              UNIX
make /f simple.mak                              MS-DOS/VMS
```

To run the program:

```
iserver -se -sb simple.b4h                      (UNIX)
iserver /se /sb simple.b4h                       (MS-DOS/VMS)
```

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.b4h                             UNIX
isim /bq simple.b4h                             MS-DOS/VMS
```

4 Programming single transputers

This chapter provides an introduction to occam programming using the toolset, using an example program for single processors. The chapter follows on from the information and example given in chapter 3 'Getting started'. For information on programming multitransputer networks see chapter 5.

Before reading this chapter the user should already be familiar with the concepts and syntax of the occam programming language. For detailed information about the language see the '*occam 2 Reference Manual*' and for an introduction to occam see '*A tutorial introduction to occam programming*'.

4.1 Program examples

A simple programming example, to get you started, is provided in section 3.3.

This chapter uses a more complex example, illustrating separate compilation; which can be found in section 4.12.

All the example programs are designed for boot from link boards. If you have a board that boots from ROM you should set it to boot from link or run the example programs using the T425 simulator tool **isim**.

4.2 occam programs

Within the toolset a single processor program is a single occam procedure with a fixed pattern of formal parameters, as illustrated below.

```
#INCLUDE "hostio.inc"
PROC occam.program (CHAN OF SP fs, ts,
                    []INT memory)
  ...  body of program
:
```

The procedure and its parameters can have any legal occam names. You must always supply the procedure with the same type of formal parameters as shown above, to enable communication with the host.

All occam procedures are terminated by a colon (:), at the same indentation as the corresponding **PROC** keyword. Do not forget the colon at the end of a program.

Program input and output is supported by the host file server, which is resident on the host computer. Access to the host file server is via the i/o libraries, which are described in part 2, chapter 1. Whenever routines from these libraries are used the channels **fs** and **ts** must be passed to the routine so that it can communicate with the host file server.

Channel **fs** comes from the host file server and **ts** goes to the host file server. Both use protocol SP, which is defined in the include file **hostio.inc**. Figure 4.1 shows how these channels are connected.

The array **memory** contains the free memory remaining on the transputer evaluation board after the program code has been loaded and the workspace allocated. It is calculated by subtracting the area occupied by the program code and data from the value specified in the **IBOARDSIZE** host environment variable. The **memory** array is passed to the program as an array of type INT, where it can be used. By allowing programs to be run on boards with different memory sizes, this array aids program portability between different boards.
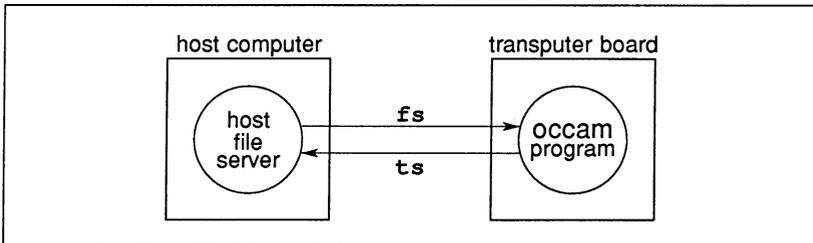
Figure 4.1 Program input/output

4.2.1 Compiling programs

The compiler produces object code in TCOFF format for input to the linker. The compiler is capable of compiling code for any one of a range of transputers (the IMS T212, M212, T222, T225, T400, T414, T425, T800, T801 or T805) in one of three error modes and with interactive debugging either enabled or disabled. The compiler enables interactive debugging by default unless the compiler '**Y**' option is used.

The standard error modes are HALT system and STOP process. A special mode, UNIVERSAL, enables code to be compiled so that it may be run in either HALT or STOP mode. The target processor and error mode must be specified for each compilation, using options on the command line. By default the compiler compiles for an IMS T414 in HALT mode, and when compiling for this transputer type and error mode you may omit the options. In all other cases the options must be supplied.

Other operating features of the compiler may be changed by options. See section 25.2 for a full description of these options.

If the compiler detects any errors, the file name and line number of each error is displayed along with a message explaining the error.

If the compilation succeeds, the compiler creates a new code file in the current directory. The filename for the new file is derived from the name of the source file and the default file extension .tco is added. The filename can also be specified on the command line.

Compilation information

It is sometimes necessary to check how much workspace (data space) will be required to run the code. This information is stored in the code file produced by the compiler, linker and librarian. To display the information use the 'I' command line option or use the binary lister tool ilist. For details of ilist see chapter 20.

4.2.2   Linking programs

When all the component parts of a program have been compiled they must be linked together to form a whole program. Component parts include the main program, any separately compiled units, and any libraries used by the program, including the compiler libraries.

If required, the compiler libraries are automatically loaded by the compiler unless specifically disabled with the compiler 'E' option. If you are unsure whether your program uses the compiler libraries it is best to always link in the appropriate library. Only library modules actually used by the compiled code will be included in the linked code file. The correct library for your program depends on the transputer type of the compilation.

To assist the user, three linker indirect files are supplied listing the compiler libraries appropriate to different processor types. The relevant file should be included on the linker command line using the 'f' option. occam2.lnk is provided for the T2 series, occam8.lnk for the T8 series and occama.lnk for other 32-bit transputers.

For further details of the compiler libraries see part 2, section 1.2.

By default, the order in which the code modules are specified on the command line determines their order within the linked unit; library modules being placed after the separately compiled modules. This default can be overruled by using the compiler directive #PRAGMA LINKAGE (see section 25.10.7) and the linkage

directive #SECTION (see section 19.3.1). These directives enable the user to prioritise the order in which modules are linked together and so influence the use of on-chip RAM. A map of the linked unit, showing the order of the modules, may be produced by specifying the linker command line option 'MO'.

4.2.3 Viewing code

Object code files produced by compiling or linking programs can be examined using the binary lister tool ilist. Information that can be displayed includes procedure definitions, exported names, external references within the code, and symbol data. For more details see chapter 20.

4.2.4 Making bootable programs

Code that has been linked to form a program cannot be loaded directly onto a transputer evaluation board, for two reasons. Firstly, object code produced by the linker and compiler tools contains extra information required by some tools. This information must be removed before the program can be loaded. Secondly, code to be run on a board which boots from link, such as the IMS B004, require the addition of bootstrap information to load the program and start it running.

Extraneous data is removed, and a boot-from-link bootstrap is added, by the collector tool icollect.

4.2.5 Loading and running programs

Bootable programs can be loaded onto the transputer evaluation board using the host file server iserver (see chapter 22).

The server must be given a number of parameters when it loads a program. All server options are two characters long, with 'S' as the first character. Server parameters are removed from the command line by the server, so you should avoid using the same options for your own program (it is best to avoid giving programs two letter options beginning with the letter 'S').

To load a program use the 'SB' option and specify the file to be loaded. This has the same effect as using options 'SR', 'SS', 'SI', and 'SC' together, that is, it resets the board, provides access to host facilities such as file access and terminal i/o, and loads the program. The 'SI' option directs the tool to display progress information as it loads the file. To terminate when the transputer error flag is set, thereby enabling the program to be debugged, add the server 'SE' option.

Programs can also be loaded onto transputer networks, without using code on the root transputer, by first using the **iskip** tool to set up a skip process and then loading the program using **iserver**. This can be useful when loading programs onto external networks via a transputer evaluation board. It is also useful for debugging programs that normally use the root transputer to run all or part of a program. The debugger always runs on the root transputer. Provided the network has at least one processor which is not used by the program, **iskip** may be used in conjunction with **iserver** to load the program over the root transputer. For details of skip loading see section 6.6.

4.3 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. The examples used in this section follow on from the example introduced in chapter 3.

4.3.1 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 straighforward. 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:

```
oc simple -T800                                    (UNIX)
ilink simple.tco hostio.lib -T800 -f occam8.lnk
```

```
oc simple /T800                                    (MS-DOS/VMS)
ilink simple.tco hostio.lib /T800 /f occam8.lnk
```

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:

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 4.1 Transputer classes and target processor

```
oc simple                                        (UNIX)
ilink simple.tco hostio.lib -f occama.lnk

oc simple                                        (MS-DOS/VMS)
ilink simple.tco hostio.lib /f occama.lnk
```

4.3.2   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 4.1 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:

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

2 Using table 4.1 select the class which may be run on all the target pro-
cessors.

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:

```
oc simple -T5                                          (UNIX)
ilink simple.tco hostio.lib -T5 -f occama.lnk

oc simple /T5                                          (MS-DOS/VMS)
ilink simple.tco hostio.lib /T5 /f occama.lnk
```

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

```
oc simple -TA                                          (UNIX)
ilink simple.tco hostio.lib -TA -f occama.lnk

oc simple /TA                                          (MS-DOS/VMS)
ilink simple.tco hostio.lib /TA /f occama.lnk
```

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.

4.3.3   Mixing code compiled for different targets

This section describes how object code compiled for one target processor or
transputer class can call and 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
  be called by 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 4.2 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 4.2 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 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, (see below).

The same rules must also be followed during the program design stage, when deciding which modules should call each other. Code for a different transputer class can be called provided that it uses the instruction set or a subset of the instruction set of the calling class. This is because the compiler needs to know which modules to select from libraries containing copies for different processor types.

Hence the headings in table 4.2 can be modified slightly to produce table 4.3 which identifies for each class the list of possible classes which it may call.

Calling class	Transputer classes which may be called
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 4.3 Calling transputer classes

In addition, the order in which the program modules are compiled is affected, in that a module which is called must be compiled before the calling module is compiled. This is explained in section 4.9 and an example is given in section 4.12.

Classes T8 and T9 cannot call or be linked with class TA; this is a change from the IMS D705/D605/D505 versions of the toolset. The reason why these classes cannot be linked together is explained in section 4.3.4, 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 4.2 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.

4.3.4   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 trans-
  puter classes support 32-bit transputers.

- Class T3 is the same as class T2 except that T3 has some extra instruc-
  tions to support CRC and bit operations and includes special debugging
  functions.

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

- 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 pro-
  cessor 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 part 2, section B.6.

- For the T414, T400 and T425 processors i.e. transputer classes T4 and
  T5 the implementation of REAL arithmetic is in software. These trans-
  puters make use of a small number of floating point support instructions
  listed in part 2, section B.5.

- 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.

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

Note: code which includes CRC, 2D block moves and floating point operations
implemented by ASM or GUY code cannot be compiled for classes TA or TB. The
compiler will report an error if this is attempted.

When considering the similarities and differences in the instruction sets of differ-
ent transputer classes it helps to divide them into the three separate structures
as shown in figure 4.2.
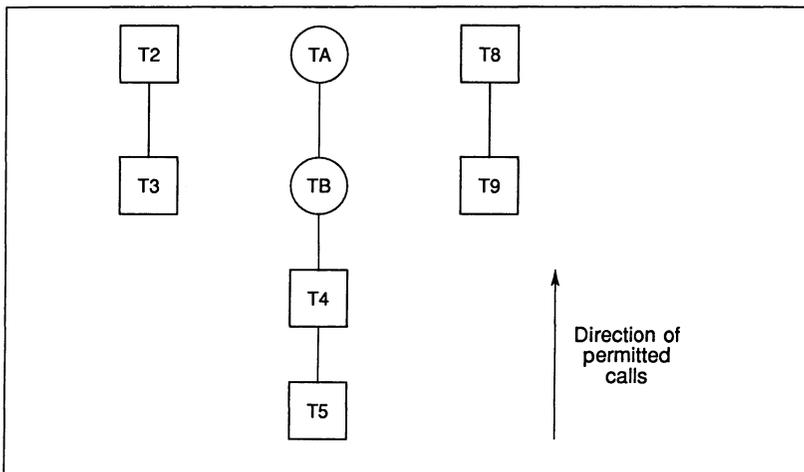
Figure 4.2 Structures for mixing transputer types and classes

By comparison with tables 4.2 and 4.3 it can be seen that a module may only call and 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 call or 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 CRC and 2D block move instructions to be used, whereas these instructions are not available to code compiled for classes TA and TB.

4.4   Error modes

For systems that require maximum security and reliability, the error behaviour is
of great concern. occam 2 specifies that run-time errors are to be handled in
one of three ways, each suitable for different programs. The error mode to be
used is supplied as a parameter to both the compiler and linker. The options are
listed in table 4.4.

Option(s)	Description
H	HALT mode
S	STOP mode
X	UNIVERSAL mode

Table 4.4 Compiler and linker options for selecting error mode

The first mode, called HALT system mode, causes all run-time errors to bring the
whole system to a halt promptly, ensuring that any errant part of the system is
prevented from corrupting any other part of the system. This mode is extremely
useful for program debugging and is suitable for any system where an error is
to be handled externally. HALT system mode is the default for the compiler, and
you should use this mode when you may want to use the debugger.

Note: on the IMS T414, T222 and M212, HALT mode does not work for pro-
cesses running at high priority, as the **HaltOnError** flag is cleared when going
to high priority.

The second mode, called STOP mode, allows more control and containment of
errors than HALT mode. This maps all errant processes into the process STOP,
again ensuring that no errant process corrupts any other part of the system. This
has the effect of gradually propagating the STOP process throughout the system.
This makes it possible for parts of the system to detect that another part has
failed, for example, by the use of 'watchdog' timers. It allows multiply-redundant,
or gracefully degrading systems, to be constructed.

The third mode, called UNIVERSAL mode, may behave as either HALT or STOP
mode depending on the transputer's Halt-On-Error flag. For example if a library is
compiled in UNIVERSAL mode, it may be linked in HALT mode with HALT mode
modules and it will behave as if it had been compiled in HALT mode. Alternatively
if it is linked in STOP mode with STOP mode modules it will behave as if it had
been compiled in STOP mode.

If a program, targetted at a single processor, is compiled and linked in UNIVER-
SAL, the collector tool will treat the linked unit as though it had been linked in
the default error mode which is HALT mode.

All separately compiled units for a single processor must be compiled and linked

with compatible error modes. Where a library is used the module of the appropriate error mode will be selected.

Code which is compiled in either HALT or STOP mode can call code compiled in UNIVERSAL mode, however code compiled in UNIVERSAL mode may only call code which has also been compiled in UNIVERSAL mode. Code which has been compiled in HALT mode may not call or be called by code compiled in STOP mode. The linker will report an error if user attempts to link HALT and STOP modules together.

4.4.1 Error detection

In some circumstances it may be desirable to omit the run time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes.

The compiler provides three command line options to enable the user to control the degree of run time error detection; they are the '**K**', '**U**' and '**NA**' options and they prevent the compiler from inserting code to explicitly perform run time checks.

These options should only be used on code which is known to be correct. The compiler does not insert a lot of error checking code so it should only be disabled as a last resort.

It is the user's responsibility to ensure that errors cannot occur. The ability to disable certain error checking code by using the '**K**' and '**U**' options should not be abused in an attempt to use illegal code, since there is no way of telling the compiler to ignore all errors.

The '**K**' option disables the insertion of code to perform run time range checking. In this context range checking only includes checks on array subscripting and array lengths. **Note**: in any situation where the compiler can detect a range check error without specifically adding code, it *may* still do so. The type of situation where this is likely to happen is when an array subscript such as $[i + j]$ is used, and $i + j$ overflows.

The '**U**' option disables the insertion of code whose only purpose is to detect some kind of error. This option is stronger than the '**K**' option, and includes the '**K**' option, so it is not necessary to use both options together. (**Note**: that the '**U**' does not include the '**NA**' option which is described below).

The '**U**' option will disable the insertion of run-time checks to detect occurrences such as the following:

negative values in replicators
errors in type conversion values,
errors in the length of shift operations,
zero length moves,
array range errors,
errors in replicated constructs such as `SEQ`, `PAR`, `IF` and `ALT`.

Note: again in any situation where the compiler can detect an error without specifically inserting code, it *may* still do so. Thus arithmetic overflows, etc, can still cause an error. (To avoid overflow errors the operators PLUS, MINUS and TIMES can be used).

If the 'ʊ' option is used in conjunction with HALT mode, it will prevent explicit checking for floating point errors in those cases where library calls are not used to perform floating point arithmetic (see below). In addition if the 'ʊ' option is used with STOP or UNIVERSAL mode, it inhibits the ability of the system to gradually propagate a STOP process throughout the system. This means that the 'ʊ' option, when used with any error mode produces identical code. The object file, however, is still marked as being compiled in a particular error mode.

Thus, faster code is produced by using the 'ʊ' option with any error mode. Any libraries which are linked with the modules will maintain the error mode and level of error detection that they were compiled for. In practice, libraries compiled in HALT mode will be fastest, so for benchmarking, modules should be compiled in HALT mode and the 'ʊ' option used.

If the user requires the equivalent of the UNIVERSAL error mode implemented by the IMS D705/D605/D505 versions of the toolset, then UNIVERSAL error mode should be used and the 'ʊ' option specified. However, the compiler will not incorporate library entries compiled with the 'ʊ' option.

The following points summarise the differences in the implementation of error detection between the current release and previous releases of the toolset i.e the IMS D705/D605/D505 toolsets.

Comparison of error modes with the IMS D705/D605/D505 toolsets

The detection of errors and the action that is taken when an error is detected are separated in the current toolset.

HALT and STOP mode behave the same as they did in the previous toolsets.

UNIVERSAL mode no longer turns error detection off, instead it produces code which may be linked in either HALT or STOP mode.

The degree of run-time checking may be reduced by using the 'K' and 'U' command line options.

To obtain the equivalent of the UNIVERSAL mode implemented by the IMS D705/D605/D505 toolsets, compile in UNIVERSAL mode and use the 'U' option. **Note**: this will not cause the compiler to incorporate libraries compiled with the 'U' option.

To obtain the equivalent of occam UNDEFINED mode (see the '*occam 2 Reference Manual*'), compile in any error mode and use the 'U' option.

The 'NA' option disables the insertion of code to check calls to ASSERT.

The occam 2 compiler recognises a procedure ASSERT with the following parameter:

PROC ASSERT (VAL BOOL test)

At compile time the compiler will check the value of test and if it is FALSE the compiler will give a compile time error; if it is TRUE, the compiler does nothing. If test cannot be checked at compile-time then the compiler will insert a run-time check to detect its status. The 'NA' option can be used to disable the insertion of this run-time check.

4.5 Interactive debugging

The compiler and linker tools support interactive debugging by default. When interactive debugging is enabled the compiler or linker will generate calls to library routines to perform channel input and output rather than using the transputer's instructions. This does cause a performance penalty to be incurred when interactive debugging is enabled. Disabling interactive debugging by using the command line option 'Y' results in faster code execution.

Interactive debugging must be enabled in order to use the interactive features of the debugger. However, the debugger does not have to be present in order to run the code.

Code which has interactive debugging disabled may call code which has inter-active debugging enabled but not vice versa. If interactive debugging is disabled for any module in a program this will prevent the whole program from being debugged interactively.

4.6   Alias and usage checking

The compiler implements the alias and usage checking rules described in the *'occam 2 Reference Manual'*. Alias checking ensures that elements are not referred to by more than one name within a section of code. Usage checking ensures that channels are used correctly for unidirectional point-to-point commu-nication, and that variables are not altered while being shared between parallel processes. For a further discussion of the rationale behind these rules, see sec-tions 25.13 and 25.14. Information is also given in *The Transputer Applications Notebook – Architecture and Software, Chapter 6 – The development of occam 2'*.

Alias and usage checking during compilation may be disabled by means of the compiler options 'A' and 'N'. Using the 'N' option it is possible to carry out alias checking without usage checking. However, it is not possible to perform usage checking without alias checking, as the usage checker relies on lack of aliasing in the program. If you switch off alias checking with option 'A', usage checking is automatically disabled.

The 'K' and 'U' options will also disable the insertion of alias checks that would otherwise be performed at run-time. These options do not affect the insertion of alias checks at compile time nor the insertion of usage checks which are only performed at compile time.

Alias checking can impose some code penalties, for example, extra code is inserted if array accesses are made which cannot be checked until runtime. The 'WO' command line option will produce a warning message every time one of these checks is generated. However, alias checking can also improve the quality of code produced, since the compiler can optimise the code if names in the program are known not to be aliased.

The compiler usage check detects illegal usage of variables and channels, for example, attempting to assign to the same variable in parallel. The compiler performs most of its checks correctly, but with certain limitations. Normally, if it is unable to implement a check exactly, it will perform a stricter check. For example, if an array element is assigned to, and its subscript cannot be evaluated at compile time, then the compiler assumes that all elements of the array are assigned to. If a correct program is rejected because the compiler is imposing too strict a rule, it is possible to switch off usage checking.

It should also be noted that usage checking can slow the compiler down. For example, programs which contain replicated constructs defined with constant values for the *base* and *count*, will be checked for each iteration of the routine. Replicated constructs which have variable *base* and *count* values are only checked once with a stricter check, because the compiler cannot evaluate, at this point, the actual limits of the replication.

4.7 Using separate vector space

The compiler normally produces code which uses separate vector space. Arrays which are declared within a compilation unit are allocated into a separate 'vector space' area of memory, rather than into workspace when they are either:

- greater than 8 bytes or

- greater than 1 word, where the elements are smaller than a word (e.g. `[5]BTYE`).

This decreases the amount of stack required, which has two benefits: firstly, the offsets of variables are smaller, access to them is faster; secondly, the total amount of stack used is smaller, allowing better use to be made of on-chip RAM.

The compiler option 'V' disables the use of a separate vector space, in which case arrays are placed in the workspace.

When a program is loaded onto a transputer in a network, memory is allocated contiguously, as shown in figure 4.3.

This allows the workspace (and possibly some of the code) to be given priority use of the on-chip RAM. Generally, the best performance will be obtained with the separate vector space enabled.

The default allocation of an array can be overridden by an allocation immediately after the declaration of an array. This allocation has one of the forms:

PLACE *name* **IN VECSPACE :**

or **PLACE** *name* **IN WORKSPACE :**

For example, in a program which is normally using the separate vector space, it may be advantageous to put an important buffer into workspace, so that it is more likely to be put into internal RAM. The program would be compiled with

```
                              MOSTNEG INT
  ┌─────────────────────────┐ + IBOARDSIZE
  │                         │
  │   Unallocated memory    │
  │  (passed as parameter   │
  │      to program)        │
  │                         │
  ├─────────────────────────┤
  │                         │
  │   occam vector space    │
  │                         │
  ├─────────────────────────┤
  │                         │
  │      occam code         │
  │                         │
  │                         │
  ├─────────────────────────┤
  │    occam workspace      │
  │                         │ MemStart
  │  Reserved by transputer │
#80000000─────────────────────┘ MOSTNEG INT
```
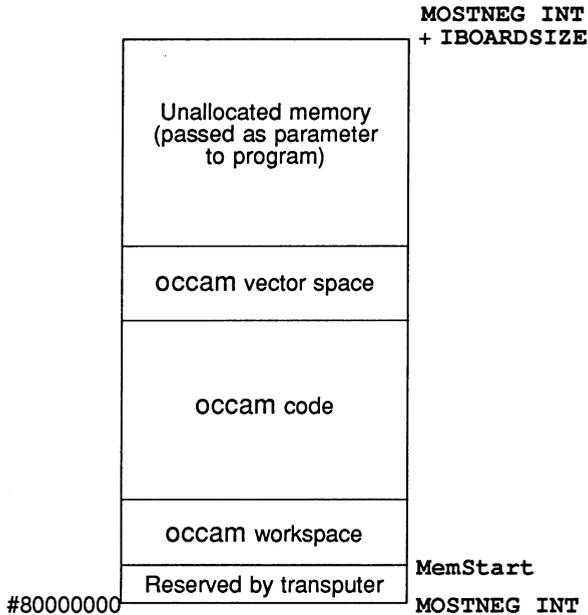
Figure 4.3 Memory allocation on a 32-bit transputer

separate vector space enabled, but would include something like:

```
[buff.size]BYTE crucial.buffer :
PLACE crucial.buffer IN WORKSPACE :
```

For a program where it is required to put all of the data apart from one large array into the workspace, the program would be compiled with separate vector space disabled, and the array allocated to vector space by a place statement such as **PLACE large.array IN VECSPACE**.

Within a program it is possible to mix code compiled with separate vector space on and code compiled with separate vector space off. The parts of the program which have been compiled with separate vector space enabled will be given use of the vector space.

Note that certain libraries such as `hostio.lib` use vector space. Therefore, it is likely that some use of vector space will be made, even if vector space is disabled for a program module.

4.8 Sharing source between files

The source of a program can be split over any number of files by using the #INCLUDE directive. This directive enables the user to specify a file which contains occam source. The contents of this file are included in the source at the same point and with the same indentation as the #INCLUDE directive. Include files may be nested to a maximum depth of twenty. By convention .inc file extension should be used for occam constant and protocol definitions. An example of using the #INCLUDE directive is given below:

#INCLUDE "infile.inc" -- source in infile.inc

The name of the file to be included is placed in quotes. All of the line following the closing quote may be used as for comments. Directives must occupy a single line.

4.9 Separate compilation

Separate compilation reflects the hierarchical structure of occam, and the occam compiler compiles occam procedures and/or functions (PROCs and FUNCTIONs). Any number of procedures and/or functions may be compiled at any time, provided the only external references they make are via their parameter lists.

A group of procedures and/or functions that are compiled together are known as a compilation unit. Each procedure and/or function in such a group may be called internally by other procedures declared later in that group, or externally by any occam in the scope of the directive which references that separate compilation unit. Constant declarations and protocols are also permitted inside a compilation unit, for the use of the procedures and functions within it. The scope of a separate compilation unit is the same as any normal occam procedure or function.

Separately compiled units are referenced from occam source as object code files, using the #USE directive. The object file may be a compiled (.tco) or library (.lib) file. If the file extension is omitted the compiler adds the extension of the current output file. This will be (.tco) unless an output file has been specified using the 'O' option.

An example of how to reference a separately compiled unit is shown below.

#USE "scunit.tco" -- code in file scunit.tco

The filename must be enclosed in double quotes. All of the line following the closing quote can be used as comment. The directive must occupy a single line.

Separate compilation units may be nested to any depth and may contain
#INCLUDE directives. They may also use libraries, as described in section 4.11.

A separate compilation unit must be compiled before the source which references
it can be compiled.

4.9.1   Sharing protocols and constants

occam constants and protocols may be declared and used within a compilation
unit according to the rules of the language. Where a constant and/or protocol is
to be used across separate compilation boundaries, it should always be placed
in a separate file. The file should be referenced in any compilation unit where it
is needed, by using the #INCLUDE directive before any #USE directive, which
introduces procedures using the protocol in their formal parameter lists. Proto-
cols will also need to be referenced in any enclosing compilation unit (because
the channels will either be declared there or passed through). For example,
suppose we have a protocol P defined in a file **myprot.inc**. We might then
use it as follows:

```
PROC main()
  #INCLUDE "myprot.inc"
  #USE "mysc.tco"

  CHAN OF P actual.channel :
  PAR
    do.it(actual.channel)
    ...
:
```

The separately compiled procedure **do.it**, in the file **mysc.occ**, would look
like this:

```
#INCLUDE "myprot.inc" -- declares protocol P
PROC do.it (CHAN OF P in)

  SEQ
    ... body of procedure
:
```

Since the protocol name P occurs in the formal parameter list of the separately
compiled procedure **do.it**, the compilation unit must include a #INCLUDE
directive, preceding the declaration of **do.it**, to introduce the name P.

4.9.2 Compiling and linking large programs

Building a program which includes separate compilation units and library references is straightforward. Separate compilation units in the program can be compiled individually by applying the compiler to them. Nested compilation units must be compiled in a bottom-up order before the top level of the program is compiled; finally the whole program is linked together.

Separate compilation units must be compiled before the unit which references them can be compiled. This is because the object code contains all the information about a unit (names, formal parameters, workspace and code size, etc.) which is needed to arrange the static allocation of workspace and to check correctness across compilation boundaries. This information may be viewed using the ilist tool.

When a program is linked the code for all the separate compilation units in the program is copied into a single file. In addition, code for any libraries used is included in the file. Where libraries contain more than one module, only those modules containing routines actually required in a program are linked into the final code. This helps to minimise the size of the linked code.

The target processor or transputer class and error mode must be specified to the linker to enable it to select appropriate library modules. Only one processor type or class may be used for the linking process and this must be compatible with the transputer type or class used to compile the modules. The error mode used for the linking process must also be compatible with the error mode(s) used to compile the modules. Compatible use of the compiler and linker 'Y' option must also be adopted for the modules to be linked.

If there are a large number of input modules, they may be supplied to the linker, within an indirect file, as a list of filenames. Indirect files may also contain directives to the linker. Linker directives enable the user to customise the linkage operation and include facilities to modify the use of workspace, create forward references to symbols and to nest indirect files. Chapter 19 provides detailed information of how to run and use the linker.

4.10   Using imakef

When a change is made to part of a program it is necessary to recompile the program to create a new code file reflecting the change. The purpose of the separate compilation system is to split up a program so that only those parts of the program which have changed or which depend on the changed units, need to be recompiled, rather than needing to recompile the whole program. However, it would be tedious to have to remember which modules had been edited, which modules might be affected by calls and the order in which the modules were

compiled and linked. For this reason a Makefile generator `imakef` is supplied
with the toolset and may be used to assist with building programs consisting of
several modules. This tool, when applied to a program (or part of a program),
compiles a list of dependencies of compilation units and uses this list to produce
a Makefile. The Makefile can be used with a suitable MAKE program to recompile
only the changed parts of a program. This ensures that compilation units will
always be recompiled where a change has made this necessary.

To use the Makefile generator you must tell it the name of the file you wish to
build. The tool can produce a Makefile for any type of file that can be built with
the toolset tools. In order for `imakef` to be able to identify file types, a different
system of file extensions must be used to that used in this chapter. The file
name rules for `imakef` are described in chapter 21 together with details of how
to use the tool.

4.11   Libraries

A library is a collection of compiled procedures and/or functions. Any number
of separately compiled units may be made into a library by using the librarian.
Separately compiled units and libraries can be added to existing libraries. Each
compilation unit is treated as a separately loadable module within a library. When
compiling or linking, only modules which are used by a program are loaded. The
rules for selective loading are described in the following section.

Libraries are referenced from occam source by the #USE directive. For exam-
ple:

>   **#USE "hostio.lib"   -- host server library**

The filename is enclosed in quotes. The rest of the line, following the closing
quote, may be used for comments. Directives must occupy a single line.

Libraries should always use a `.lib` file extension, and this must always be
supplied in a #USE directive.

4.11.1   Selective loading

Each module (separately compiled unit) in a library is selectively loadable by the
linker; i.e. parts of a library not used or unusable by a program are ignored.
The unit of selectivity is the library module; i.e. if one procedure or function of a
library module is used then all the code for that module is loaded.

The compiler is selective when a library is referenced. Only modules of a library
that are of the same, or compatible, transputer type or class, error mode and

method of channel input/output, are read (see sections 4.3, 4.4 and 4.5).

Selective loading is based on the following rules:

1 The transputer type or class of a library module must be the same as, or compatible with, the code which could use it.

2 The error mode of the library module must be the same as, or compatible with, the code which could use it.

3 The interactive debugging mode (i.e. whether interactive debugging is enabled or not) of the library must be the same, or compatible with, the code which could use it.

4 At least one routine (entry point) in a module is called by the code.

Rules 1 to 3 apply to the compiler. All the rules are used by the linker. The compiler only selects on transputer type, error mode and method of channel input/output. It is not until the linking stage that unused modules are rejected. For details on mixing processor classes and error modes see sections 4.3 and 4.4 respectively.

4.11.2 Building libraries

Libraries are built using the librarian tool **ilibr**. Libraries can be created from either separately compiled units (**.tco** or library files **.lib**) or from linked units (**.lku** files) but not a combination of both. The librarian takes any number of input files and combines them into a single library file. Each separately compiled unit forms a single module in the library.

When forming a library the librarian will warn if there are multiply defined routines (entry points). In other words, for each combination of transputer type, error mode and method of channel input/output there may only be one routine with a particular name. For further information on building libraries see chapter 18.

As an example consider building a library called **mylib.lib**. The source of this library is contained in a file called **mylib.occ** and has been written to be compilable for both 16 and 32 bit transputers. We want the library to be available for T212 and T800 processors in halt on error mode only. Having compiled the source for the two processors we will have two files, for example: **mylib.t2h** and **mylib.t8h**. To form a library from these compilation units use the following command line:

```
ilibr mylib.t2h mylib.t8h
```

When an output filename is not specified, as in this example, the librarian uses the first file in the list to make up the output file name and adds the extension `.lib`. In this case it will write the library to the file `mylib.lib`.

The librarian can also take an indirect file containing a list of the files to be built into the library. Such files should have the same name as the library, but with a `.lbb` file extension. So, still using the above example, if the names of the files to make up the library were put in a file called `mylib.lbb`, we could then build the library using one of the following commands:

```
ilibr -f mylib.lbb -o mylib.lib          (UNIX)
ilibr /f mylib.lbb /o mylib.lib          (MS-DOS/VMS)
```

Compiled modules can be added to an existing library file. However, if the librarian attempts to create an output file with the same name as an input library file, an error will be produced. This can be avoided by specifying a different output filename using the 'o' option. Alternatively if one on the compiled modules to be added to the library has a different name, this could be specified first on the command line. Once the new library file has been created it can be renamed if necessary. Adding modules to an existing library does not require programs which call it, to be recompiled, provided it is given its original name in its final form.

The Makefile generator `imakef` can be used to assist with the building of libraries. This is particularly useful where libraries are nested within other libraries or compilation units, because `imakef` can identify the dependencies of libraries on other modules or separately compiled units. For further information about the `imakef` tool see chapter 21.

For further details of how to use the librarian and how to optimise libraries see chapter 18.

4.12   Example program – the pipeline sorter

This section introduces an example which serves to show how a large program might be structured, in terms of separate compilation units, libraries, and a shared protocol.

4.12.1   Overview of the program

The program sorts a series of characters into the order of their ASCII code values.

Figure 4.4 shows the basic structure of this program. There are three processes:
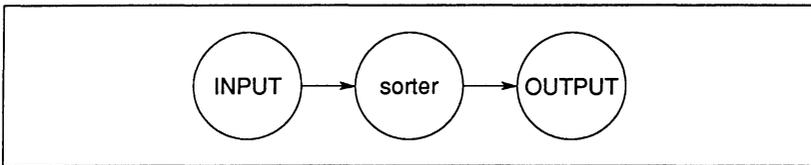
Figure 4.4 Basic structure of sorter program

the input process, the output process and the sorter process. We can decompose the sorter process by using a pipeline structure. This uses the algorithm described in *'A tutorial introduction to occam programming'*. If we design the pipeline carefully we can ensure that each element of the pipeline is identical to all the other elements. The pipeline is served by an input process, which reads characters from the keyboard, and an output process which writes the sorted characters to the screen. Figure 4.5 shows the structure of the program using a pipeline.
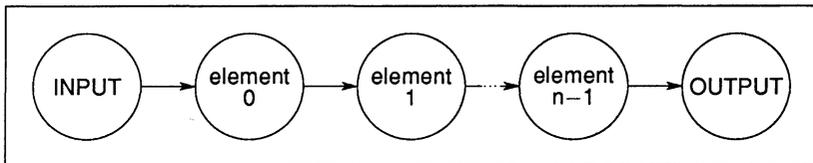
Figure 4.5 Pipeline of n elements

An obvious implementation would be to write an occam process for each process in figure 4.5, using a replicated process for the pipeline. Communication between the processes is via occam channels and to aid program correctness we should use an occam PROTOCOL for these channels. This protocol must be shared by all the processes. As the occam compiler compiles processes (PROCs) and as each of the processes is independent we can implement each one as a separately compiled unit. The processes share a common protocol and the best way to ensure consistency is to place the protocol in a separate file and use the #INCLUDE mechanism to access it. These processes can then be called in parallel by an enclosing program which can access the code of each process by the #USE mechanism.

There is a problem with this implementation because two processes require access to the host file server. The host file server is accessed via a pair of occam channels and occam does not allow the sharing of channels between processes. There are a number of ways around this problem. One solution is to use a multiplexor process for the server channels, as described in section 8.5. Another solution is to merge the two processes into a single process. This solution is used because the program accesses the server in a sequential manner (read a line then display sorted line, read a line etc.). Figure 4.6 gives the final

process diagram for the program.

INPUT/OUTPUT → element 0 ········· → element n−1
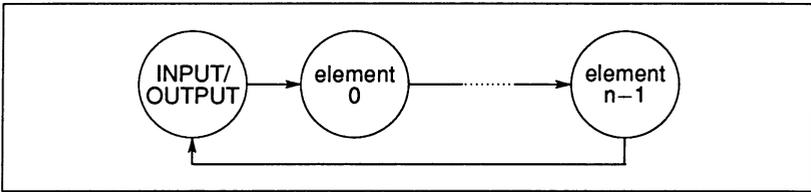
Figure 4.6 Program with combined input/output process

The implementation can be split into four files:

element.occ   the pipeline sorting element

inout.occ   the input/output process

sorter.occ   the enclosing program

sorthdr.inc   the common protocol definition

Figure 4.7 shows the way these files are connected together to form a program.

```
              sorter
  #INCLUDE            #USE
             #USE
           element    inout
       #INCLUDE
              #INCLUDE
   sorthdr
```
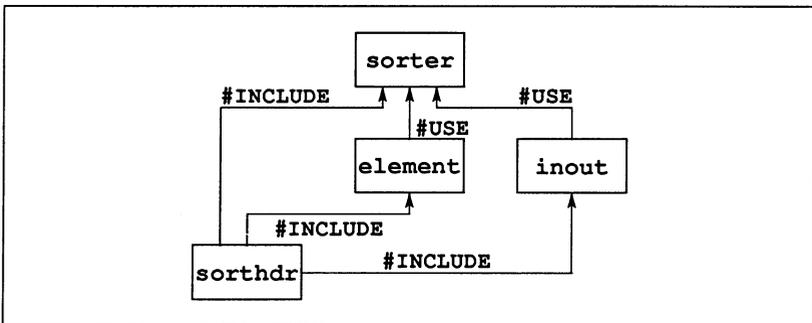
Figure 4.7 File structure of program

The source of the program is given below and is supplied in the 'examples' directory. You can either copy these files to a working directory or you can type in the source as given below. For details of the toolset directories see the Delivery Manual that accompanies the shipment.

Two other files are required to complete the program. These are the host file server library **hostio.lib** and the corresponding **.inc** file containing the host file server constants.

4.12.2   The protocol

Declarations of constants and channel protocols are contained in the include file `sorthdr.inc`, which is listed below.

```
PROTOCOL LETTERS
  CASE
    letter; BYTE
    end.of.letters
    terminate
:


VAL number.elements IS 100:
```

This declares a protocol called `LETTERS`, which permits three different types of message to be communicated:

`letter`   – followed by the character to be sorted.

`end.of.letters`   – marks the end of the sequence to be sorted.

`terminate`   – signals the end of the program.

The constant `number.elements` is also declared. This defines both the number of sorting elements in the pipeline and the maximum length of the sequence of characters that can be sorted.

4.12.3   The sorting element

The sorting element `element.occ` is listed below:

```
#INCLUDE "sorthdr.inc"

PROC sort.element (CHAN OF LETTERS input, output)

  BYTE highest:
  BOOL going:

  SEQ
    going := TRUE
    WHILE going
      input ? CASE
        terminate
          going := FALSE
```

```
letter; highest
  BYTE next:
  BOOL inline:
  SEQ
    inline := TRUE
    WHILE inline
      input ? CASE
        letter; next
          IF
            next > highest
              SEQ
                output ! letter; highest
                highest := next
            TRUE
              output ! letter; next
        end.of.letters
          SEQ
            inline := FALSE
            output ! letter; highest
  output ! end.of.letters
  output ! terminate
:
```

This program consists of two loops, one nested inside the other. The outer loop accepts either a termination signal or a character sequence for sorting. If it receives a character it enters the inner loop. The inner loop reads characters until it receives an 'end of letters' signal, signifying the end of the string of characters to be sorted. The sort is performed by storing the highest (ASCII) value character it receives and passing any lesser (or equal) characters on to the next process. The 'end of letters' tag causes the stored value to be passed on and the inner loop terminates.

The maximum number of characters which can be sorted is determined by the number of sorter processes. One character is sorted per process.

4.12.4 The input/output process

This process consists of a loop which reads a line from the keyboard, then sends the line to the sorter and, in parallel, reads the sorted line back. It then displays the sorted line. If the line read from the keyboard is empty the loop is terminated. At the end of the process the host file server is terminated with the success constant `sps.success`, which is defined in the file `hostio.inc`.

If any i/o errors occur the program will stop, allowing it to be examined by the debugger.

The input/output process inout.occ is listed below.

```
#INCLUDE "sorthdr.inc"
#INCLUDE "hostio.inc"

PROC inout (CHAN OF SP fs, ts,
            CHAN OF LETTERS from.pipe, to.pipe)

  #USE "hostio.lib"

  [number.elements - 1]BYTE line, sorted.line:
  INT line.length, sorted.length:
  BYTE result:
  BOOL going:

  SEQ
    so.write.string.nl (fs, ts,
    "Enter lines of text to be sorted *
    *- empty line terminates")
    going := TRUE
    WHILE going
      SEQ
        so.read.echo.line(fs, ts, line.length,
                          line, result)
        IF
          result <> spr.ok
            STOP -- stop if an error occurs
          TRUE
            so.write.nl (fs, ts)
        PAR
          SEQ
            IF
              (line.length = 0)    -- no more input
                to.pipe ! terminate
              TRUE
                SEQ
                  SEQ i = 0 FOR line.length
                    to.pipe ! letter; line[i]
                  to.pipe ! end.of.letters
          BOOL end.of.line:
          SEQ
            end.of.line := FALSE
            sorted.length := 0
            WHILE NOT end.of.line
              from.pipe ? CASE
                terminate
```

```
                        SEQ
                          end.of.line := TRUE
                          going := FALSE
                     letter; sorted.line[sorted.length]
                          sorted.length := sorted.length + 1
                     end.of.letters
                        SEQ
                          so.write.string.nl(fs, ts,
                            [sorted.line FROM 0
                             FOR sorted.length])
                          end.of.line := TRUE
          so.exit(fs, ts, sps.success) -- terminate server
:
```

4.12.5   The calling program

This process calls the input output process in parallel with the sorter elements, in a pipeline. The **memory** parameter must be declared, but the program does not use it.

The calling program **sorter.occ** is listed below.

```
#INCLUDE "hostio.inc"

PROC sorter (CHAN OF SP fs, ts, []INT memory)

  #USE "hostio.lib"  -- host i/o library
  #INCLUDE "sorthdr.inc"

  #USE "inout"       -- separately compiled units
  #USE "element"

  [number.elements + 1]CHAN OF LETTERS pipe:
  PAR    -- run pipe between i/o processes
    inout(fs, ts, pipe[number.elements], pipe[0])
    PAR i = 0 FOR number.elements
      sort.element(pipe[i], pipe[i + 1])
:
```

4.12.6   Building the program

To build the program, first compile each component of the program separately, link them together, and add bootstrap code to the main compilation unit.

The program's components must be compiled in a bottom up fashion, that is, `element.occ` and `inout.occ` first (in either sequence), followed by the main program `sorter.occ`.

First, compile the sorting element `element.occ` using the following command:

```
oc element
```

The file extension can be omitted on the command line because the source file has the conventional extension `.occ`.

The compiler produces a file called `element.tco`, compiled for a T414 in HALT mode.

Compile the input/output process using the following command:

```
oc inout
```

The compiler will produce a file called `inout.tco`, compiled for a T414 in HALT mode.

Then compile the main body using the command line:

```
oc sorter
```

The compiler will produce a file called `sorter.tco`, compiled for a T414 in HALT mode.

Having compiled all the components of the program you can now link them together to form a whole program. Any libraries used by the program must also be specified to the linker. The library `hostio.lib` is the server library used by this program. Remember the include file, `occama.lnk`, which identifies the other libraries, such as compiler libraries, required in the linking process. (See section 4.2.2). To link the files use one of the following commands:

```
ilink sorter.tco inout.tco element.tco hostio.lib -f occama.lnk
ilink sorter.tco inout.tco element.tco hostio.lib /f occama.lnk
```

When specifying options for any of the tools remember to use the correct prefix character for your version of the toolset ('−' for UNIX implementations, and '/' for the IBM PC and VAX/VMS implementations).

The linker will create the file `sorter.lku` linked for a T414 in HALT mode.

If a main entry point is not specified, the linker uses the first valid entry point that it encounters in the input. Therefore, in the above example, it is important

to list the file '`sorter.tco`' first. A main entry point may be specified within an indirect file using the linker directive `#mainentry` or on the command line using the '`ME`' option.

Before you can run the program you must add bootstrap code. To do this use the collector tool `icollect`, using one of the following command lines:

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

The '`t`' option informs the collector tool that the input file is a linked unit rather than the output of the configurer tool. (The configurer is used for multi-processor applications).

The collector tool will create the files `sorter.btl` and `sorter.cfb`. The `.btl` file contains the bootable program code. The `.cfb` file is a configuration binary file which is created by `icollect` as a by-product of creating the bootable file; it is redundant as far as this example is concerned.

To run the program on a transputer board use one of the following commands:

```
iserver -se -sb sorter.btl                (UNIX)
iserver /se /sb sorter.btl           (MS-DOS/VMS)
```

The '`sb`' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The '`se`' option directs the server to terminate if the program sets the error flag. For more details about the server options see chapter 22.

The program reads characters from the keyboard, sorts the line and redisplays it. The program will run until input is terminated by typing RETURN on an empty line.

Figure 4.8 shows an example of the screen display, obtained by running `sorter.btl` on a UNIX based toolset. The user inputs the string 'Sorter program ' and terminates the program by pressing RETURN.

```
iserver -se -sb sorter.btl

Enter lines of text to be sorted - empty line terminates

Sorter program
Saegmooprrrrt
```

Figure 4.8 Example output produced by running `sorter.btl`.

To run the program using the simulator use one of the following commands:

```
isim -bq sorter.btl                          (UNIX)
isim /bq sorter.btl                     (MS-DOS/VMS)
```

The 'bq' option specifies batch quiet mode which causes the simulator to run the program and then terminate. For more details about how to use the simulator see chapter 23.

4.12.7   Automated program building

The `imakef` tool can be used to automate the development process. From the above example it can be seen that there are many steps to go through when building a program of any size. Some of these steps must be performed in a specific order and if part of the program were changed then all affected parts must be recompiled and relinked etc.

MAKE is a common tool for building programs. It uses information about when files were last updated, and performs all the necessary operations to keep object and bootable files up to date with changes in any part of the source. Makefiles are the standard method of providing the MAKE program with the information it needs.

The occam toolset is designed in such a way that it is possible for a tool to construct Makefiles to build occam programs. The Makefile generator `imakef` produces Makefiles in a format acceptable to most MAKE programs.

`imakef` requires the user to adopt a particular convention of file extensions. The user then only has to specify the target file he requires i.e. a bootable file and `imakef`, using its knowledge of file names rules, creates a suitable Makefile. This file has full instructions on how to build the program.

By running the MAKE program for the file the entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

For more details about the `imakef` tool and an example of how to create a makefile for the pipeline sorter program used in this chapter, see chapter 21.

5 Configuring transputer networks

This chapter describes how to build programs that run on networks of transputers. It describes how to configure an occam program for a network of transputers using the occam configurer tool `occonf` and describes how to load the program onto a transputer network. These procedures are illustrated with an example program for four transputers.

The chapter introduces the configuration language, whose syntax is specified in part2, appendix E and the configurer tool `occonf`, described in chapter 26. This chapter also includes examples illustrating various aspects of configuration.

5.1   Introduction

In order to build programs for multitransputer networks a program is split into a number of self contained components, and each of these is implemented as an occam process. Each process may communicate with other processes resident on the same transputer or, via links, with processes on other transputers.

Programs consisting of occam processes can be run on single or multiple transputers, in any combination. Performance requirements can be met by adapting the application to run on differing numbers of transputers, and by using differing network topologies. The mapping of processes to processors on a transputer network is known as configuration.

Transputer programs can be configured to run on any physical network of transputers. They can be configured to be loaded from an external host down a transputer link, or to be loaded from ROM.

Configuration is achieved by including the program in a configuration description written in the occam configuration language. A configuration description is created by the user as a text file using the configuration language which is an extension of occam. The file is expected by `occonf` to have the file extension `.pgm`. A configuration description may be processed by the configurer tool to generate a configuration data file, which in turn may be processed by the collector tool `icollect` to generate a transputer loadable file.

Conventional file name extensions may be used for these various file types to facilitate the construction of Makefiles using the Makefile generator tool. Chapter 21 describes how to use the Makefile generator for program development and the extensions which should be used.

Within a configuration description the hardware network and the software description are kept separate. This enables the software description to be used for running the same parallel program on a variety of alternative hardware networks. Likewise a particular physical network may be described once for use in a variety of configurations describing different programs that may be run on the same network.

By using the facilities for calling other languages from occam, programs compiled from mixed language sources may also be configured using the occam configurer. (These facilities enable the foreign language code to be incorporated into the occam program as equivalent occam processes. An example of this is provided in the **examples** directory supplied with the toolset. A description of this method of mixed language programming is given in *ANSI C toolset user manual*). Similarly it is possible to configure occam modules (which are called by C programs) using the configurer provided with the ANSI C toolset. Details of how to do this are also given in the *ANSI C toolset user manual*.

5.2   Configuration model

The configuration model consists of the following parts:

- A hardware network description which declares a network as a connected graph of processors.

- A software description in the form of an occam process.

- A mapping between the processes and channels of the software and the nodes (processors) and arcs (transputer link connections) of the network. The mapping is achieved by declaring names and, in the scopes of these declarations, referring to the names in the structures of the configuration description. Normal occam scope rules apply.

The software description takes the form of an occam process with at least as many parallel sub-processes as there are hardware processors in the network. Within the description, each process which may be independently placed on a processor, is introduced by a **PROCESSOR** construct naming a processor. Processors so named may either be the hardware processors declared in the network description, or may be logical processors mapped onto the hardware processors in a separate mapping structure. In either case the processor name must have appeared in a **NODE** declaration in whose scope the software description is written.

The connections between processes in the software description are defined by occam channels. It is thus possible for the configurer tool to determine what code is to be loaded onto what processor, and to choose its own mapping of

channels onto physical connections between processors.

Some channels may be used to connect to hardware outside the network, such as the development host or other hardware connected by means of link adaptors. External objects of this kind are declared as EDGEs in the hardware description.

All processors which are connected together are connected via their links, represented in the language as attributes, of type EDGE of declared NODEs.

The connections to external edges, or those between processors may optionally be declared as ARCs, which associate a name with a particular connection. This enables explicit mappings of channels onto these arcs to be made.

5.2.1 Configuration language

A configuration description consists of a sequence of declarations and statements in an extension to occam and follows the usual occam scope rules. These declarations and statements are evaluated by the occam compiler, which is called during configuration by the configurer tool occonf. Appendix E (in part 2) defines the syntax of the occam configuration language and also gives details of how it differs from previous implementations of the toolset i.e. the IMS D705/D605/D505 products.

Configuration declarations introduce physical processors, arcs and edges of the network, network connections and processor attributes, logical processors to be mapped onto physical processors, the software description, and the mapping between logical and physical processors.

Arrays of NODEs, EDGEs, and ARCs may also be declared. A configuration description includes one NETWORK, one CONFIG and, optionally, one MAPPING. Each of the items appearing before CONFIG behaves as an occam specification, and ordinary VAL abbreviations may be included amongst these components to facilitate the description of scalable configurations. A NETWORK, CONFIG or MAPPING is optionally named by an identifier following its opening keyword.

Configuration declarations are usually followed by statements which perform various actions relating to the declaration. Actions are defined by SET, CONNECT and MAP statements. The DO construct enables these statements to be grouped or replicated. PROCESSOR statements introduce processes which may be mapped onto named processors.

The MAP statement may be replicated, via the DO construct, within a MAPPING declaration. SET and CONNECT statements may be used within a NETWORK declaration and may be combined in any order using the DO construct.

Declaration	Description
NODE	Introduces processors (*nodes* of a graph). These processors are considered to be *physical* if they are defined as part of the hardware description, or *logical* if they are defined as part of the software description and mapped to a physical processor as part of the mapping.
ARC	Introduces named connections (*arcs* of a graph) between processors (using the transputer links). These connections need not be declared as ARCs unless channels are required to be explicitly placed on particular links.
EDGE	Introduces external connections of the hardware description. External edges may be the host, or any peripheral connected via a link adaptor e.g. a joystick, disc drive.
NETWORK	Defines the connections and attribute settings of previously declared NODEs (physical processors).
MAPPING	Defines mappings between logical processors and physical processors.
CONFIG	Introduces the software description.

Table 5.1 Configuration description declarations

Statement	Description
SET	Defines values for NODE attributes.
CONNECT	Defines a connection between two EDGEs, either of two nodes or between a node and a declared external EDGE.
MAP	Defines the mapping of a logical processor onto a physical processor declared as a NODE.
PROCESSOR	Introduces a software process and associates it with a logical or physical processor.
DO	Groups one or more actions defined by SET, CONNECT or MAP statements.

Table 5.2 Configuration description statements

Code from other files may be referenced by means of the #USE directive, either at the top level, or within the CONFIG construct. #INCLUDE directives can be used to include other source files.

It is suggested that the distinct sections are kept in different files, accessed by #INCLUDE directives from a 'master' file.

5.2.2   Overall structure of a configuration description

A configuration description consists of two or three parts; a hardware network description, a software network description, and an optional mapping between the two.

The hardware description defines processor connections. It also defines attributes such as processor types and memory sizes. These processors are known as *physical* processors.

The software description is basically an occam parallel process, annotated with PROCESSOR statements to indicate which processes are to be compiled for which processors. These processes are allocated to *logical* processors.

The mapping section can be used to ease the task of changing a particular program to execute on a different hardware network. The mapping section enables this to be performed without modifying the software description in any way, by flexibly mapping the *logical* processors onto the *physical* processors. As an optimisation, for simple programs, or for programs which will never need to be re-mapped, the software description may reference the *physical* processors directly, avoiding the need to introduce *logical* processor names.

The following example illustrates the basic style of the language:

```
-- hardware description, omitting host connection
VAL K IS  1024 :        -- useful constants for memory
VAL M IS K * K :        -- sizes

NODE root.p, worker.p :    -- declare two processors
NETWORK  simple.network
  DO
    SET root.p (type, memsize := "T414", 1 * M)
    SET worker.p (type, memsize := "T800", 4 * M)
    CONNECT root.p[link][3] TO worker.p[link][0]
:
-- mapping
NODE root.l, worker.l :
MAPPING
  DO
    MAP root.l ONTO root.p
    MAP worker.l ONTO worker.p
:

-- software description
#INCLUDE "prots.inc" -- declare protocol
#USE "root.lku"      -- must be linked units
```

```
#USE "worker.lku"
CONFIG
  CHAN OF protocol root.to.worker, worker.to.root :
  PLACED PAR
    PROCESSOR root.l
      root.process(worker.to.root, root.to.worker)
    PROCESSOR worker.l
      worker.process(root.to.worker, worker.to.root)
:
```

Note that the configurer can, in this example, automatically place the channels onto the single connecting link, assuming that the two channels are used in different directions. The configurer can make this check by means of the normal occam usage checking rules.

This example is illustrated in figure 5.1.
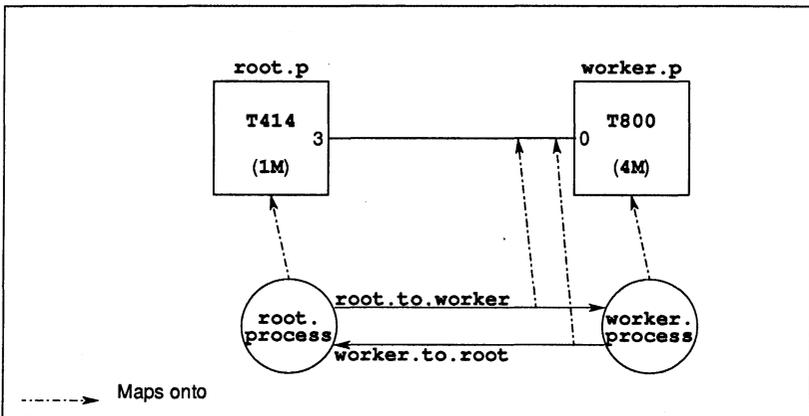
Figure 5.1 Mapping of software onto hardware

In a simple configuration such as this one where each physical processor is mapped onto a single logical processor, a shortened configuration description may be used which omits the mapping section altogether and uses the physical processor names directly in the software description.

To devise this shortened description remove the mapping section and delete the suffixes .p and .l from the NODE declarations, SET, CONNECT and PROCESSOR statements.

5.3 Hardware description

5.3.1 Declaring processors

Processors are declared to have NODE type, as if they were occam data items:

```
NODE worker :                    -- single processor
[No.of.workers]NODE pipeline : -- array of processors
```

5.3.2 NODE attributes

A NODE has a set of attributes, analogous to fields of a record. An attribute is referenced by subscripting the name of the node with the name of the attribute. The attributes are:

```
[]BYTE type :      -- String describing processor type,
                   -- see list below
[]EDGE link :      -- Link connections, number may
                   -- depend on type
INT memsize :      -- Memory size in BYTEs
BOOL root    :      -- Defines root processor if there is
                        no HOST connection
INT romsize :      -- Size of ROM attached to processor
order.code  :      -- Defines the priority of the program
                        code in memory
order.vs    :      -- Defines the priority of the
                        program's vectorspace in memory
```

The list of permissible attributes is in general dependent upon the NODE type field, and may be extended for other NODE types in the future.

The attribute names, which are predeclared by the configurer, do not follow the occam scope rules; they are only recognised in the correct context.

The use of order.code and order.vs is explained in section 5.5.3.

5.3.3 NETWORK description

The NETWORK keyword introduces a section which describes the connectivity, and attributes of previously declared NODEs. These should be declared outside of the NETWORK description, so that they are visible inside and below the NETWORK description.

To describe a single processor, the SET statement provides values for the pro-

cessor's attributes in the style of a multiple assignment.

```
NETWORK single
  SET processor ( type, memsize := "T800", 1024*1024)
:
```

The **type** attribute must be set to a **BYTE** array (of any length) whose contents describe the processor type. Trailing spaces at the end of the processor's type are ignored.

Supported types are:

```
    "T212"   "T222"   "T225"   "M212"
    "T400"   "T414"   "T425"
    "T800"   "T801"   "T805"
```

The **memsize** attribute must be set to the amount of usable memory attached to that processor, as a contiguous amount starting at the most negative address. It is specified in **BYTES**.

Both the **type** and **memsize** attributes must be defined for all processors. No attribute may be defined more than once for each processor.

The above example could also be written as a sequence of **SET** statements in a DO construct:

```
NETWORK single
  DO
    SET processor ( type    := "T800")
    SET processor ( memsize := 1024*1024)
:
```

Since the DO construct does not imply any particular ordering, there is no constraint on the order in which attributes may be defined.

If a network is to be configured to be loaded from ROM, the attribute **root** must be set to **TRUE** for one processor only. By default this attribute is **FALSE** for all processors. The attribute **romsize** should be set to the number of bytes of ROM on the root processor. These attributes are ignored if the network is configured to be booted from link.

IF, **SKIP** and **STOP** may be used in DO constructs and are effectively executed at configuration time.

Processors must be connected together by means of **CONNECT** statements quot-

ing a pair of edges:

```
VAL K IS 1024:
NETWORK pair.from.ROM
  DO
    SET proc1 ( type, memsize := "T800", 2048 * K)
    SET proc1 ( root, romsize := TRUE, 256 * K)
    SET proc2 ( type, memsize := "T414", 1024 * K)
    CONNECT proc1[link][0] TO proc2[link][3]
:
```

The order of the two edges in a CONNECT statement is irrelevant.

Arrays of processors do not need to all have the same types or attributes. They can be set by using DO replicators within the NETWORK construct, and by using conditionals, as in this (rather contrived) example:

```
NETWORK pipe
  DO
    DO i = 0 FOR 100
      IF
        (i \ 4) = 0
          SET processor[i] (type, memsize := "T800",
                                  4 * (1024 * 1024) )
        TRUE
          SET processor[i] (type, memsize := "T414",
                                  2 * (1024 * 1024) )

    DO i = 0 FOR 99
      DO
        CONNECT processor[i][link][1] TO
                processor[i+1][link][0]
        IF
          (i \ 2) = 0
            CONNECT processor[i][link][2] TO
                    processor[i+2][link][3]
          TRUE
            SKIP

:
```

More complicated expressions may also be used, as long as they can be evaluated at configuration time:

```
VAL processors IS ["T414", "T414", "T414", "T800"] :
NETWORK fancy   -- every fourth processor is different!
  DO i = 0 FOR SIZE array
    SET array[i] ( type := processors[i \ 4] )
:
```

5.3.4   Declaring EDGEs

Declared EDGEs define the ends of external connections of a NETWORK. For
instance, a connection to another machine whose internal structure is irrelevant.
They are declared as though they were occam data types, and as usual we
can declare arrays of them:

```
[10]EDGE diskdrive :
NETWORK disk.farm
  DO i = 0 FOR 10
    DO
      -- insert code to set attributes, then:
      CONNECT processor[i][link][0] TO diskdrive[i]
:

EDGE joystick :
NODE controller :
NETWORK n
  DO
    SET controller (type, memsize := "T212", 64 * 1024)
    CONNECT controller[link][2] TO joystick
:
```

5.3.5   Declaring ARCs

In some circumstances a programmer may require to name a connection be-
tween two processors. This isn't normally necessary, because the configurer
can place channels between processors onto links automatically, but where a
channel must be connected onto an external EDGE this is required. Also, if
there are multiple links between two processors, and one link is set for some
reason to go at a different data rate than another, the programmer might wish to
have more control.

These named links are called ARCs, and are declared as though they were
occam data types. They are associated with a link connection by adding a
WITH clause to the end of a CONNECT statement.

```
EDGE joystick :
ARC  link.to.joystick :
NODE controller :
NETWORK n
  DO
    SET controller (type, memsize := "T212", 64 * 1024)
    CONNECT controller[link][2] TO joystick WITH
                                       link.to.joystick
  :
```

5.3.6   Abbreviations

occam style abbreviations are permitted, to enable easier reference to elements of arrays, etc:

```
[10]NODE pipe :
NETWORK pipeline
  DO i = 0 FOR 10
    NODE this IS pipe[i] :
    SET  this (type, memsize := "T414", 1024*1024)
  :
```

Since NODEs have an attribute link, whose type is []EDGE, we can abbreviate one link of a processor as an EDGE:

```
[10]NODE pipe :
NETWORK pipeline
  DO
    DO i = 0 FOR 10
      SET pipe[i] (type, memsize := "T414", 1024*1024)
    DO i = 0 FOR 9
      EDGE this IS pipe[i  ][link][2] :
      EDGE that IS pipe[i+1][link][3] :
      CONNECT this TO that
  :
```

Simple one-to-one mappings of logical to physical processors may also be expressed as abbreviations:

```
NODE root.l IS root.p :
```

5.3.7   Host connection

There is a predefined EDGE named HOST, which indicates the connection to a host computer:

```
NODE single :
ARC  hostlink :
NETWORK B004
  DO
    SET single (type, memsize := "T800", 1000000)
    CONNECT single[link][0] TO HOST WITH hostlink
:
```

When configuring a program which is designed to be booted via a transputer link, one processor *must* be connected to the predefined EDGE HOST.

5.3.8   Examples of network descriptions

1) *Single processor configuration connected to host:*

```
NODE MyB004:
ARC hostlink:
NETWORK B004
  DO
    SET MyB004 (type, memsize := "T414", 2 * M)
    CONNECT MyB004[link][0] TO HOST WITH hostlink
:
```

This configuration is illustrated in figure 5.2.
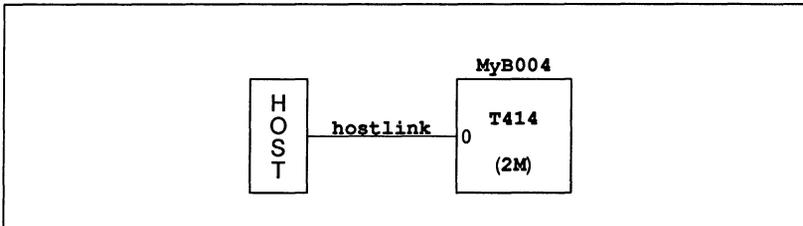
Figure 5.2 Example of host connection

2) *Simple pipe with one processor with different memory size:*

```
[p]NODE Pipe:
ARC hostLink:
NETWORK simple.pipe
  DO
    CONNECT HOST TO Pipe[0][link][0] WITH hostLink
    DO i = 0 FOR p-1
      CONNECT Pipe[i][link][2] TO Pipe[i+1][link][1]
    SET Pipe[0] (type, memory := "T800", 2*M)
    DO i = 1 FOR p
      SET Pipe[i] (type, memory := "T800", 1*M)
  :
```

This network is illustrated in figure 5.3.
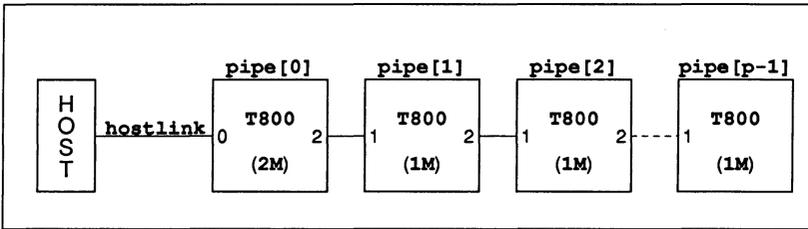
Figure 5.3 Simple pipeline with different processor memory sizes

3) *Square array with host interface processor:*

```
VAL Up IS 0:
VAL Left IS 1:
VAL Down IS 2:
VAL Right IS 3:
NODE HostSquare:
[p][p]NODE Square:
ARC hostlink:
NETWORK square
  DO
    SET HostSquare (type, memsize := "T414", 2*M)
    CONNECT HOST TO HostSquare[link][0] WITH hostlink
    CONNECT HostSquare[link][1] TO
            Square[p-1][p-1][link][Down]

    DO i = 0 for p
      DO j = 0 for p
        DO
          SET Square[i][j] (type, memsize := "T800", 1*M)
          IF
```

```
(i = 0) AND (j = 0)
  CONNECT HostSquare   [link][Down] TO
          Square[0][0][link][Up]
  i = 0
    CONNECT Square[p - 1][j - 1][link][Down] TO
            Square[0     ][j    ][link][Up]
  TRUE
    CONNECT Square[i - 1][j][link][Down] TO
            Square[i    ][j][link][Up]

DO i = 0 for p
  DO j = 0 for p
    IF
      j = (p-1)
        CONNECT Square[i][j][link][Right] TO
                Square[(i + 1)\p][0][link][Left]
      TRUE
        CONNECT Square[i][j][link][Right] TO
                Square[i][j + 1][link][Left]
:
```

5.4   Software description

The software description is an occam process, **PAR** or **PLACED PAR**, with pro-
cesses annotated by **PROCESSOR** statements. These identify which processes
may be placed on particular processors. The keyword **PLACED** is retained for
compatibility with earlier products; it is no longer required and has no effect.

The **NODE**s which are referenced by a **PROCESSOR** statement may be either
physical processors if they are described as part of the hardware description, or
logical processors if they are described as part of the software description. If
the latter, they are mapped onto physical processors by means of a **MAPPING**
section.

Physical processor names are allowed here to simplify small networks, or those
which will not be re-mapped, so that the programmer does not need to invent
two names for each processor.

The *logical* processor names must be introduced first by means of **NODE** declara-
tions. These look identical to those used in the hardware description, but cannot
have attribute settings. Since these must be visible to a following **MAPPING**
section, they must be declared *outside* the **CONFIG** construct. Channels which
are to be placed on **ARC**s by mapping statements must also be declared outside
the **CONFIG** construct.

The process 'inside' the **PROCESSOR** statement may consist of occam text.

However, it is recommended that the code should be restricted to simple procedure calls i.e. to separately compiled procedures, referenced as linked compilation units using the #USE directive. Code which generates library calls is not allowed.

A PROCESSOR statement associates the process instance (*process*) it labels with the logical or physical processor it names. The same name may be referenced in more than one PROCESSOR statement. The set of processes so named will run in parallel on that processor.

Note: when imakef is used to build the program, any linked units referenced by the software description must be given extensions of the type *cxx*. This is because imakef uses a different convention for file extensions to the normal TCOFF file extensions, see chapter 21.

5.4.1   Libraries of linked units

The facility to create libraries of linked units provides an easy method of targetting a process at different processor types within a software description.

For example, suppose a process is compiled and linked once for a T2 and once for a T8 and the linked units are given imakef file extensions in order to distinguish them. Referencing the two linked units directly within the software description by #USE directives, will cause one of them to hide the other from the configurer.

If, however, the linked units are used to create a library and this is referenced by a single #USE directive, the configurer will be able to extract the correct copy of the process for each PROCESSOR statement it finds.

Only libraries containing linked units may be referenced from within a software description.

5.4.2   Example

The following example of a software description, is for the pipeline sorter program introduced in chapter 4. The example is developed to show the complete configuration description for the program, in section 5.6. Figure 5.4 illustrates the mapping of the software processes onto a network of logical processors, which in this example is achieved without an actual mapping section. This method of mapping is explained in section 5.5.4.

```
#INCLUDE "hostio.inc"  -- declares SP
#INCLUDE "sorthdr.inc" -- declares LETTERS
```

```
#USE "inout.lku"        -- linked unit
#USE "element.lku"      -- linked unit
NODE inout.p :          -- logical processor
[string.length]NODE pipe.element.p : -- logical
                                     -- processors
CONFIG
  CHAN OF SP app.in:
  CHAN OF SP app.out:
  PLACE app.in, app.out ON hostlink:
  [string.length+1]CHAN OF LETTERS pipe:
  PAR
    PROCESSOR inout.p
      inout (app.in, app.out, pipe[string.length],
           pipe[0])
    PAR i = 0 FOR string.length
      PROCESSOR pipe.element.p[i]
        sort.element (pipe[i], pipe[i+1])
:
```

This example names a single processes **inout.p** and an array of processes **pipe.element.p**. The code may be mapped onto any hardware configuration onto which these logical processors may be mapped and which includes an **ARC** declaration for the host connection **hostlink**.
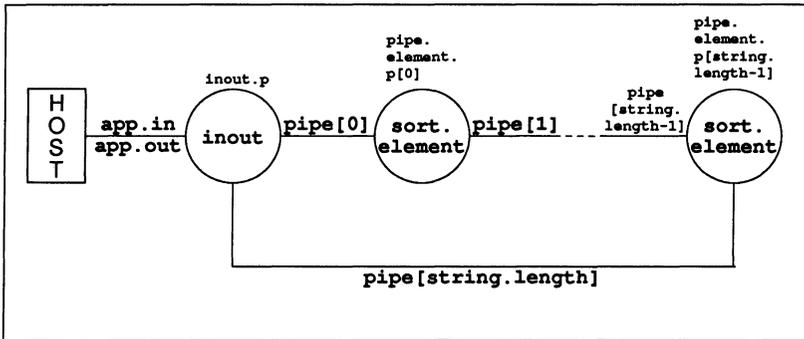
Figure 5.4 Pipeline sorter – mapping processes onto processors

5.5   Mapping descriptions

A **MAPPING** structure is used if the user has declared logical processors. The **MAPPING** maps logical processors used in the software description onto physical processors used in the hardware description. It is possible to map any number of logical processors onto any physical processor.

The priority at which a process runs may be determined as part of the mapping, if that logical process does not explicitly include high priority code. This reflects the fact that changes in mapping may not affect the overall structure of the software, but can often change the decisions made about which processes should be prioritised.

`IF`, `SKIP` and `STOP` may be used in a mapping structure.

As would be expected from the occam scoping rules, logical processor names must be declared as `NODE`s in the software description, before the opening keyword `MAPPING` of the mapping description. Each name so declared must appear once and once only on the left hand side of a mapping item. Physical processors may appear on the right hand sides of multiple mapping items.

The mapping structure itself may appear either before or after the software description.

5.5.1 Mapping processes

Having declared *physical* processors, as part of the hardware description, and *logical* processors, as part of the software description, we can assign logical processors to physical processors using the `MAP` statement.

```
MAPPING map
  MAP logical.proc ONTO physical.proc
:
```

We can also supply a list of logical processors to all be mapped onto the same physical processor:

```
MAPPING map
  MAP router.proc, application.proc ONTO root.processor
:
```

This is exactly equivalent to:

```
MAPPING map
  DO
    MAP router.proc      ONTO root.processor
    MAP application.proc ONTO root.processor
:
```

And we can use `DO` replicators, and `IF` constructs, etc:

```
MAPPING map
```

```
DO
  DO i = 0 FOR  10
    MAP router.proc[i] ONTO router.processor[i]
  DO i = 0 FOR  5
    MAP sieve.proc[i] ONTO sieve.processor
:
```

If we require that the process's priority be determined by the mapping, we can
use the optional `PRI` clause. The argument to `PRI` can be either 0 to indicate
high priority, or 1 to indicate *low* priority:

```
MAPPING map
  DO i = 0 FOR  10
    MAP logical.proc[i] ONTO physical.proc
                             PRI (INT (i = 0))
:
```

The configuration tool will reject the mapping at high priority of a process which
itself includes a `PRI PAR`.

5.5.2   Mapping channels

Channels between processors need not be placed by the user. The configurer
will determine that a connection exists, and will allocate all the channels to links
if they are available. However, if a user wants to override the default allocation,
channels may be mapped onto named `ARC`s. Also, channels connecting pro-
cessors to external `EDGE`s must be mapped onto an `ARC` which connects to that
`EDGE`.

Channels are mapped onto `ARC`s in exactly the same way as logical processors
are mapped onto physical processors. Two channels may be mapped onto the
same `ARC`, as long as they are used in different directions (the configurer will
check this). Obviously the `ARC` must connect `EDGE`s of the processors onto
which are mapped the processes which use the channel.

```
EDGE peripheral :
ARC  peripheral.arc :
NODE root.proc :
NETWORK n
  DO
    -- insert code to set attributes, then:
    CONNECT root.proc[link][0] TO peripheral WITH
                               peripheral.arc
:
CHAN OF protocol to.periph, from.periph :
```

```
NODE process :
CONFIG
  PLACED PAR
    PROCESSOR process
      -- reads from channel from.periph, writes to
      -- channel to.periph
:

MAPPING
  DO
    MAP process ONTO root.proc
    MAP to.periph, from.periph ONTO peripheral.arc
:
```

5.5.3 Moving code and data areas

Two processor attributes may be used to provide greater control of the layout of code and data areas in memory. Note that changing the default ordering means that the INMOS debugger cannot be used with the program, and for this reason these attributes must be explicitly enabled on the command line by means of the 'RE' option.

Normally the configurer arranges for the program's workspace to be given the highest priority, and hence placed at the lowest address on chip. This means that the workspace can make best use of the transputer's on-chip RAM. Program code is treated with next priority, and vectorspace has the lowest priority.

These priorities can be overridden by setting two processor attributes: 'order.code' and 'order.vs', which correspond to the program code, and to the program's vectorspace, respectively. These can be set to INT values, where lower integers indicate a higher priority. The workspace is given priority 0. Hence setting 'order.code' to -1 means that the code will be placed at a lower address than the workspace. If an attribute is not set, the priority is considered to have value 0. The relative ordering of sections whose priorities are equal is undefined.

Since these attributes are essentially properties of the user's program, not of the hardware description, the settings must be made as part of the MAPPING section. However, the processor which is referenced must be a physical processor.

Thus we may have a mapping section like so:

```
MAPPING prioritise.code
  DO
    SET physical.processor (order.code := -1)
    MAP logical.processor ONTO physical.processor
  :
```

If code re-ordering has not been explicitly enabled by the command line option 'RE', these attributes will be ignored.

5.5.4 Mapping without a MAPPING section

Without a mapping section a channel allocation may be used instead of a channel mapping.

Any channel in scope at the point where a process is labelled is available for explicit *placement* on an arc declared in the hardware network. This is done by adding the following *allocation* immediately after the declaration of the channel:

```
CHAN OF protocol to.periph, from.periph :
PLACE to.periph, from.periph ON peripheral.arc :
CONFIG
  PLACED PAR
    PROCESSOR root.proc
    -- as before
  :
```

Allowing more than one channel to be placed in a single allocation or mapping statement allows the two channels on any one physical transputer link to be placed in a single line of code.

5.5.5 Mapping examples

1) *pipeline sorter on a single processor*

```
MAPPING
  DO
    MAP inout.p ONTO MyB004
    DO i = 0 FOR string.length
      MAP pipe.element.p[i] ONTO MyB004
  :
```

2) *pipeline sorter on a ring of processors, one per process*

```
MAPPING
  DO
    MAP inout.p ONTO MyB004
    DO i = 0 FOR string.length
      MAP pipe.element.p[i] ONTO ring[i]
  :
```

5.6   Example: A pipeline sorter on four transputers

This section describes how the pipeline sorter program, described in section 4.12, may be distributed over four T414 transputers. Each processor has many processes allocated to it.

An example of how to design and write a configuration description is given, followed by detailed instructions about how to compile, configure and run the program.

In the configuration description it is assumed that there is a transputer network of four T414 transputers connected as shown in figure 5.5. It does not matter if you don't have such a network – you should read through this example and then try modifying it for your network.
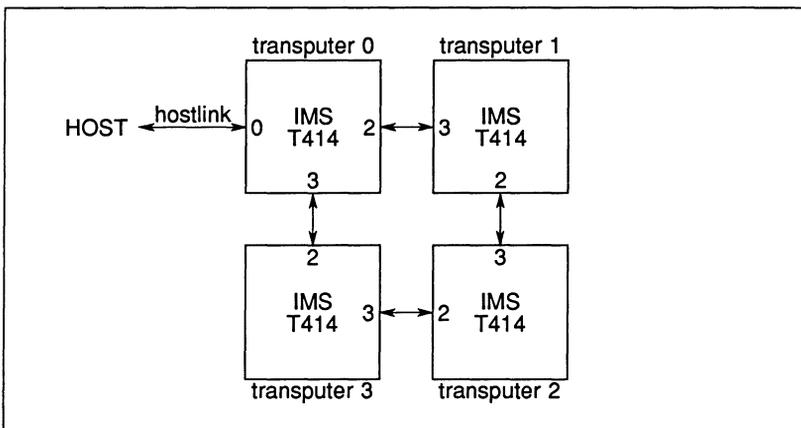
Figure 5.5 Network of four transputers

The occam source and configuration description developed in this example is supplied with the toolset in the "examples" directory, and you should copy these files to a working directory in order to build the program. Alternatively you can

type in the source of the program, as it is given below and in section 4.12.

The files are as follows:

sorthdr.inc   the common protocol definition.

element.occ   the sorting element.

inout.occ   the interface to the host file server.

sortb3.pgm   the configuration description for the network.

The contents of the files **sorthdr.inc**, **element.occ** and **inout.occ** are described in section 4.12. The contents of the other files used in the program are described below.

To complete the program the host file server library **hostio.lib**, the hostio include file **hostio.inc**, and the compiler library code will be used from the toolset library directory.

The following code is in the file **sortb3.pgm**, it describes the hardware network shown above and a mapping of processes onto this network which puts an equal number of processes on all processors after the first one, which also gets any remainder:

```
-- problem size
VAL string.length IS 80:

-- hardware description
VAL number.of.transputers IS 4:
VAL number.of.elements IS string.length:
VAL elements.per.transputer IS number.of.elements/
                                number.of.transputers:
VAL remaining.elements IS number.of.elements\
                          number.of.transputers:
VAL elements.on.root IS elements.per.transputer +
                        remaining.elements:

VAL K IS 1024:
[4]NODE B003.t:
ARC hostlink:
NETWORK
  DO
    CONNECT B003.t[0][link][0] TO HOST WITH hostlink
    DO i = 0 FOR 4
      DO
        SET B003.t[i] (type, memsize := "T414", 256*K)
        CONNECT B003.t[i][link][2] TO
```

```
                      B003.t[(i+1)\4][link][3]
:
-- mapping
VAL HIGH IS 0:   -- priorities
VAL LOW IS 1:
NODE inout.p:
[number.of.elements]NODE pipe.element.p:
MAPPING
  DO
    MAP inout.p,
        pipe.element.p[elements.on.root-1] ONTO
        B003.t[0] PRI HIGH
    DO i = 0 FOR elements.on.root-1
      MAP pipe.element.p[i] ONTO B003.t[0] PRI LOW
    DO j = 0 FOR number.of.transputers - 1
      VAL first.element.here IS elements.on.root +
          (j*elements.per.transputer):
      VAL last.element.here IS first.element.here +
          (elements.per.transputer-1):
      DO
        MAP pipe.element.p[first.element.here],
            pipe.element.p[last.element.here] ONTO
            B003.t[j+1] PRI HIGH
        DO i = first.element.here + 1 FOR
                 elements.per.transputer - 2
          MAP pipe.element.p[i] ONTO
              B003.t[j+1] PRI LOW
:
#INCLUDE "hostio.inc"
#INCLUDE "sorthdr.inc"
#USE "inout.lku"
#USE "element.lku"
CONFIG
  CHAN OF SP app.in:
  CHAN OF SP app.out:
  PLACE app.in, app.out ON hostlink:
  [string.length+1]CHAN OF LETTERS pipe:
  PAR
    PROCESSOR inout.p
      inout (app.in, app.out, pipe[string.length],
             pipe[0])
    PAR i = 0 FOR string.length
      PROCESSOR pipe.element.p[i]
        sort.element (pipe[i], pipe[i+1])
:
```

In the mapping structure shown, the logical processors named in the software description are mapped onto the physical processors declared in the hardware description. **Note**: that on each processor, processes which communicate on external channels are mapped to be run at high priority. The allocation of processes to transputers is shown in figure 5.6.
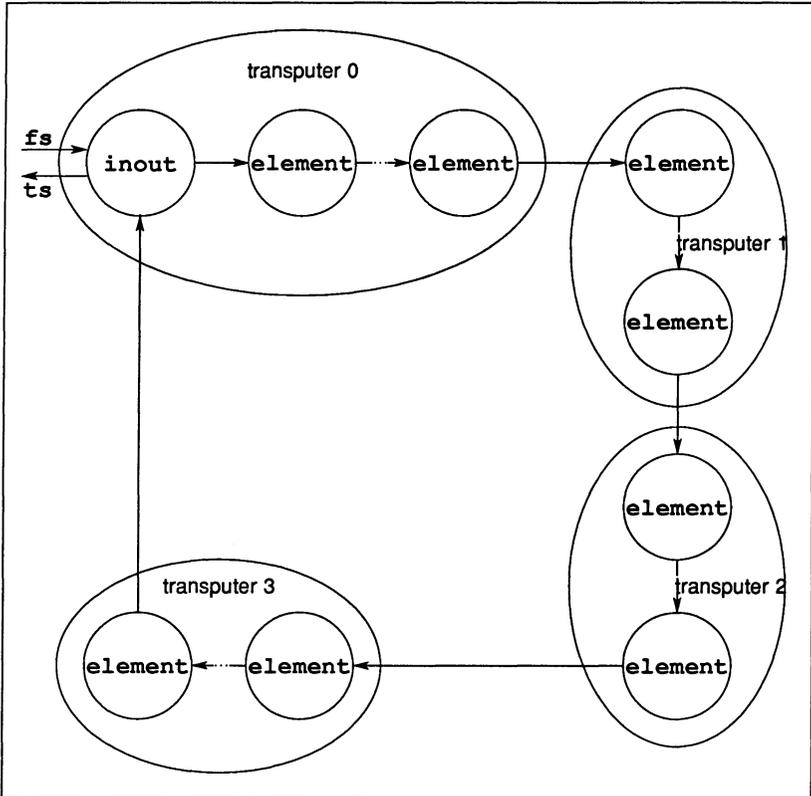
Figure 5.6 Pipeline sorter processes

5.6.1   Building the program

The components of the program must be compiled in a bottom up fashion. First compile the sorting element using the following command:

```
oc element
```

Because the file has a `.occ` file extension you can omit the extension from the filename. The command line options to specify the target processor and error mode may also be omitted because the defaults are required i.e. `T414` and HALT mode. The compiler will produce a file called `element.tco`.

Next compile the input/output process using the following command:

`oc inout` (creates the file `inout.tco`)

Each of these files must now be linked. The files are linked in separate operations, together with any files they reference. Each linking operation creates a unit of code which may be loaded onto the transputer network, according to configuration defined in the configuration description.

To link `element.tco` use one of the following commands:

```
ilink element.tco -f occama.lnk          (UNIX)
ilink element.tco /f occama.lnk      (MS-DOS/VMS)
```

Both of these commands will create a file called `element.lku`. The linker indirect file `occama.lnk` contains the necessary references to the compiler libraries. This file is supplied with the toolset.

To link `inout.tco` use one of the following commands:

```
ilink inout.tco hostio.lib -f occama.lnk          (UNIX)
ilink inout.tco hostio.lib /f occama.lnk      (MS-DOS/VMS)
```

Both of these commands will create a file called `inout.lku`.

Now configure the file `sortb3.pgm` which defines both the communication channels between the processes and how they should be loaded onto the network:

`occonf sortb3.pgm`

This command will create an output file called `sortb3.cfb`

To make the program runnable you must add bootstrap code. To do this use the collector tool `icollect`:

`icollect sortb3.cfb`

The collector will create the file `sortb3.btl`

5.6.2   Running the program

The program in the file `sortb3.btl` may be loaded and run using the skip loader from the host via the root transputer which is assumed to be connected by its link 2 to link 0 of the first transputer of the IMS B003 external network.

One of the following command sequences should be used:

UNIX based toolsets:

```
iskip 2 -e -r
iserver -se -ss -sc sortb3.btl
```

MS-DOS and VMS based toolsets:

```
iskip 2 /e /r
iserver /se /ss /sc sortb3.btl
```

To run the program on the transputer network which includes the root transputer, use one of the following commands:

```
iserver -se -sb sortb3.btl                      (UNIX)
iserver /se /sb sortb3.btl                  (MS-DOS/VMS)
```

The program will run until you type 'RETURN' on its own. The '**se**' option directs the server to terminate if the program sets the error flag.

5.6.3   Automated program building

As with the single processor version of this program it is possible to automate the building of this program with the Makefile generator tool and a suitable MAKE program. The version of the configuration program supplied in the file `sortb3c.pgm` is written using `imakef` file naming conventions. For example, the linked units are given file extensions of the form *cxx*.

To produce a Makefile for the entire program type:

```
imakef sortb3c.btl
```

The Makefile generator will produce a file called `sortb3c.mak` containing a MAKE description for the program. It will also produce linker indirect files for the two compiled units which comprise the program; these will refer to any necessary modules from the library.

To build the program run the MAKE program on the file `sortb3c.mak` and

all the necessary compiling, linking and configuration will be done automatically. For more information about MAKE programs see chapter 21.

5.7 Use of conditionals in a configuration

Conditional constructs (IF) are permitted inside **NETWORK**, **MAPPING** and **CONFIG** constructs. This makes it possible to create configuration descriptions which can be 'conditionally compiled' for different network structures.

For example, while developing a program, it may be useful to modify a program to bypass the root processor, so that an application may be placed directly onto an application processor. The following, rather trivial, example demonstrates this:

5.7.1 Example: Configuration using conditional IF

In this example, when a single processor is in use, the application communicates directly with the host, as shown in figure 5.7. When two processors are available, a buffer process is loaded onto the root processor. This process buffers the communication between the application and the host. See figure 5.8.
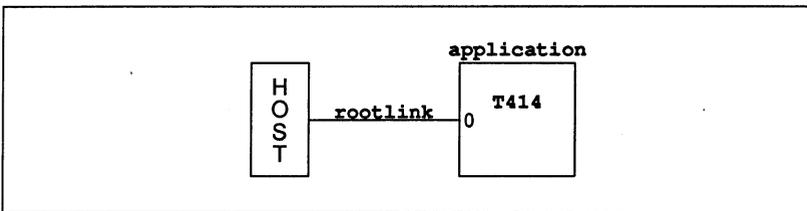
Figure 5.7 Direct host connection
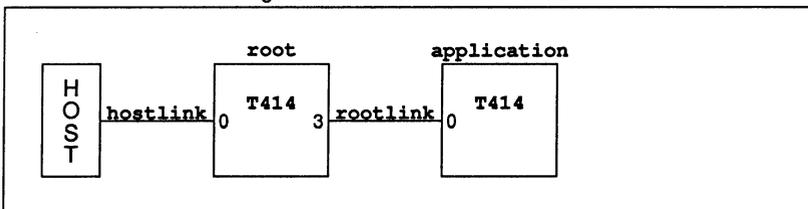
Figure 5.8 Communication via the root processor

The implementation is split into the following files:

app.occ – the application
buff.occ – the buffer process

myprog.pgm – the configuration description file

The content of app.occ is as follows:

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"

PROC application.process(CHAN OF SP fs, ts)
  SEQ
    so.write.string.nl(fs, ts, "Hello world")
    so.exit           (fs, ts, sps.success)
:
```

The content of buff.occ is as follows:

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"

PROC buffer.process(CHAN OF SP fs, ts, from.app, to.app)
  CHAN OF BOOL stopper :
  -- This never terminates
  so.buffer(fs, ts, from.app, to.app, stopper)
:
```

The content of myprog.pgm is as follows:

```
VAL number.of.processors IS 1 :  -- 1 when running,
                                 -- 2 for developing
NODE root, application :
ARC hostlink, rootlink :
NETWORK
  DO
    IF
      number.of.processors = 2
        DO
          SET root (type, memsize := "T414", #100000)
          CONNECT root[link][0] TO HOST WITH hostlink
          CONNECT root[link][3] TO application[link][0]
                                  WITH rootlink
      TRUE
          CONNECT application[link][0] TO HOST WITH rootlink
    SET application(type, memsize := "T414", #100000)
:
#INCLUDE "hostio.inc"
#USE "app.cah"
#USE "buff.cah"
CONFIG
  CHAN OF SP fs, ts :
```

```
PLACE fs, ts ON rootlink : -- Note that this is 'rootlink'
                          -- not 'hostlink'
PAR
  IF
    number.of.processors = 2
      CHAN OF SP fs0, ts0 :
      PLACE fs0, ts0 ON hostlink :
      PROCESSOR root
        buffer.process(fs0, ts0, ts, fs)
    TRUE
      SKIP
  PROCESSOR application
    application.process(fs, ts)
:
```

NODEs which are declared, but do not have any attributes set, are ignored when configuring a program.

5.8 Summary of configuration steps

To summarise, the steps involved in building a program that runs on a network of transputers are as follows:

1 Decide how your program will be distributed over the transputers in your network.

2 Write a configuration description for your program by:

  (a) Describing your hardware network.

  (b) Inserting **PROCESSOR** statements into your program and adding any necessary mapping description.

3 Compile all the separate compilation procedures that form the code for each transputer in a bottom up fashion.

4 Link each configuration procedure with its component parts into a file with the name used in **#USE** directives in the configuration source file.

5 Run the configurer on the configuration description file.

6 Collect the code using `icollect`.

7 Load the program into the network using the host file server.

Steps 3 to 6 can be automated by using `imakef` and a suitable MAKE program.

6 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.

6.1 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). **Note** a bootable file is constructed such that copying it to a link will boot the network automatically.

6.2 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.

6.3   The boot from link loading mechanism

`iserver` loads programs onto transputer networks, via the host link connection, using the communication protocol `SP`.

The bootstrap code for the transputers in the network is sent first. The code is propagated throughout the network as individual processors load neighbouring processors. After all the transputers in the network have been booted, program code is allocated to individual processors. For a multitransputer network the allocation of processes to processors is determined by the configuration file. For single transputer programs code is loaded onto the first processor on the network.

If `iskip` is used the first transputer in the network is bypassed. Therefore the network must contain one additional transputer to the number required to run the program.

When the code is copied into the transputer's memory the process boots automatically and the program continues to run until an error occurs, the server is terminated by pressing the ISERVER interrupt key (usually CTRL-C or CTRL-BREAK), or the program terminates naturally. (**Note**: terminating the server will only stop the program if the program attempts to communicate with the server).

6.3.1   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 14.3.6.

6.4   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 TRAM 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, such as that generated by `icollect`.

Examples of boot from link boards supplied by INMOS are the IMS B008 PC motherboard (with appropriate TRAMs) and the IMS B014 and IMS B016 VME-bus standard interface boards.

Boot from ROM TRAM 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.

6.4.1   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 transputer to which the subsystem is attached.

6.4.2   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 the host 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. (**Note** some boards do not conform to this system of signal propagation – see section 6.5.1).

6.5   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 14.3.

6.5.1   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 'A'

debugger option must be supplied on the debugger command line to reset the network.

6.5.2   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 `iskip` and `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 14.2.

6.5.3   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 14.3.

6.6   Example skip load

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

6.6.1   Target network

The program to be loaded is configured for a target network consisting of two T800 processors mounted on a B008 motherboard. A T414 processor in slot zero acts as the root transputer, and the target network is connected to link 2 on the root transputer via one of the links on processor 1. The two T800 processors are connected by a single link.

The target network and its connections are shown schematically below.
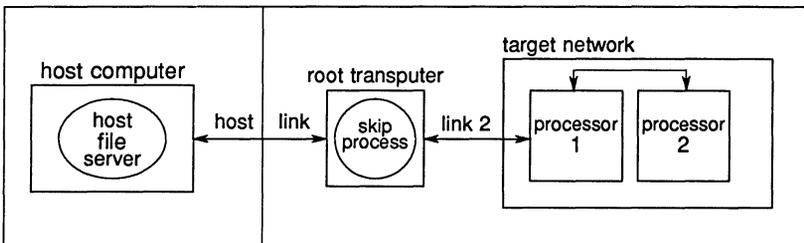
6.6.2   Loading the program

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                           (UNIX)
iskip 2 /r /e                    (MS-DOS and VMS)
```

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)
```

6.6.3 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 14.3.6.

7 Debugging occam programs

This chapter describes how to debug occam transputer programs. It describes the facilities of the toolset debugger `idebug` and shows how they can be used to debug transputer programs in a systematic manner. It explains how the debugger can be used in two modes (post-mortem and interactive) to analyse transputer programs and describes the two debugging environments (symbolic and Monitor page). The chapter ends with a tutorial example to illustrate breakpoint debugging, some hints about debugging occam code, and a list of points to note when using the debugger.

Chapter 14 provides detailed information about `idebug`, including command line syntax, symbolic debugging functions and monitor page commands.

7.1   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.

Post-mortem debugging allows programs to be examined for the cause of failure after the transputer halts 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.

7.1.1   Debugging with isim

The transputer simulator tool isim can also be used to debug transputer pro-
grams from a low level environment. Using a similar environment to the de-
bugger 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 7.12). Details of how to use the simulator tool can be found in chapter 23.

7.2   Programs that can be debugged

The debugger can analyse programs running on transputers that are either di-
rectly 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 with at least 1 Megabyte of
memory at the root (2 Megabytes or more is preferable). 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 in parallel
with the user's program.

7.3   Runtime errors

A running occam program may halt for a number of reasons. Among the most
common causes of error are:

- A **STOP** process, or a process which behaves like **STOP** (such as an **IF**
  with no **TRUE** guards) has been executed.

- An array index is out of range.

- An arithmetic error, such as overflow or divide-by-zero has occurred.

- An array element is being aliased at runtime, that is, being referred to by
  more than one name within a given scope.

When a runtime error occurs, the debugger can be used to pinpoint the line of
occam causing the error, and to investigate the state of that process and other
processes in the system. It can also be used to investigate the state of the
processor or network when the program stopped.

Note: The debugger may not find all current processes; for example, it cannot
automatically find processes which have deadlocked waiting for communication
on internal channels. Deadlocks are discussed in more detail in section 7.17.

Possible causes of runtime errors are:

STOP In occam the **STOP** process behaves as though an error has occurred.
The following occam statements behave like **STOP**:

- **IF** statements where no guard evaluates TRUE.

- **CASE** statements where no case evaluates TRUE and there is no
  ELSE statement.

- **ALT** statements where no guard evaluates TRUE.

Arithmetic errors Arithmetic errors such as overflow and divide by zero cause
an error.

Floating-point calculations cause an error if any input is infinity or 'Not-a-
Number', or if a result would be infinity or 'Not-a-Number'. This can be
avoided by explicit use of the IEEE library routines. See the 'occam 2
Reference Manual' for details.

Shifts Shifting an integer by more than the number of bits in its representation
or by a negative value causes an error.

Type conversions Type conversions where the value is not in the range ac-
cepted by the new type cause an error. For example, a value converted
to type **BYTE** must lie in the range 0–255.

Replicators Negative replicators in replicated constructs (**SEQ**, **PAR**, **IF**, or
ALT) cause an error. (Zero replicators are permitted.)

Array accesses Any access to elements outside the range of an array cause
an error. This also applies to segments of arrays.

If a segment of an array is assigned to another segment of the same
array, the two segments must not overlap.

The sizes of an array must correspond when an array is passed as a
parameter to a procedure or function, or when an array is assigned or
abbreviated. Zero length segments are allowed.

Abbreviations Abbreviating the same element of an array twice in the same
scope generates an error. The compiler 'alias checking' option '**A**' dis-
ables this form of error checking.

Communications Attempting to communicate a zero length array on a channel
of type **CHAN OF ANY** causes an error. Zero length *counted arrays* are
permitted.

A `CASE` input process where the communicated tag does not match any of those supplied, causes an error.

Retyping Any `RETYPES` expression must be aligned to the correct word or byte boundary. For example, bytes with indexes 5, 6, 7 and 8 of a declared `BYTE` array cannot be retyped as `INT32`, since `INT32`s must be aligned on a word boundary.

7.4   Compiling programs for debugging

Programs to be debugged must be compiled with full debugging data enabled; this is a default of the occam compiler.

7.4.1   Symbolic debug information

The occam compiler generates object files containing full debugging information, by default. Two command line options may be used to limit the debugging information produced by the compiler.

The '`Y`' option disables interactive debugging using breakpoints, while the '`D`' option makes the compiler produce minimal debug information only. Minimal debug information enables the debugger to backtrace out of a procedure or function to a module compiled with full debug information. It is intended for modules that are placed in libraries (e.g. the libraries supplied with this toolset are compiled with this option).

The '`D`' option only affects the debug information produced and does not alter the code generated. Code generated using the '`D`' option is identical to that generated with full debug information.

The '`Y`' option produces object code which is optimal for channel communications on a transputer. It disables channel communications via library routines (see sections 25.7 and 7.6.1). As a result, the object code produced for channel communications will often be different.

7.4.2   Error modes

Programs to be debugged should be compiled and linked in HALT mode i.e the toolset default. The behaviour of a program when an error occurs depends on the mode in which the program was compiled and linked, as follows:

- In HALT mode any error during program execution halts the transputer immediately.

- In STOP mode, errors do not halt the program, rather they stop the process allowing other processes executing on the same transputer to continue. Programs compiled in this mode can only be debugged if they are halted explicitly.

- Programs compiled in UNIVERSAL mode will adopt the error mode selected at link time i.e. HALT or STOP mode. If UNIVERSAL mode is selected at both compile and link time, then the error behaviour will default to HALT mode.

7.5   Post-mortem debugging

Post-mortem debugging is 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 compiled in HALT mode so that the processor halts when the flag is set, and they should be loaded by `iserver`, using the 'SE' option, so that the error flag is monitored.

Post-mortem debugging can also be used to debug programs that have been explicitly interrupted with 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.

7.5.1   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 15.

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 24.

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.

```
┌─────────────────────────────┬─────────────────────────────┐
│ Without debugging│           │ With debugging              │
│       kernel     │  0        │     kernel     │ 0          │
│                  │           │            3 ╱Runtime╲ 1    │
│      3│  User  │1            │             ╱ kernel  ╲     │
│       │ process │            │           2 ╲         ╱     │
│                  │           │              ╲User ╱        │
│              2   Transputer  │              ╲process╱      │
│                              │                  Transputer │
└─────────────────────────────┴─────────────────────────────┘
```
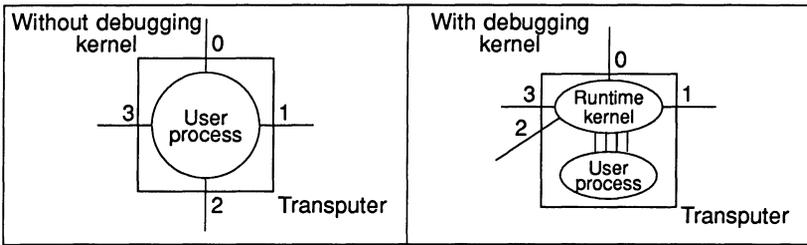
Figure 7.1 Debugger runtime kernel

7.6   Breakpoint debugging

Breakpoint debugging allows programs to executed under interactive control us-
ing 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 vari-
ables, jump down channels to other processes or processors, and determine
the state of the network. Special symbolic functions and Monitor page com-
mands, 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.

Programs that communicate to the host **must** use `iserver` SP protocol, as
used by INMOS libraries.

7.6.1   Runtime kernel

The breakpoint debugger places a special runtime kernel on each processor in
addition to the application bootable code. This kernel provides a 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 7.1.

Note: The debugging kernel places the transputer into Halt-On-Error mode re-
gardless 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. Kernels on processors
with hardware support require slightly more memory because they retain more
state information. The size of the kernel on each transputer type is given in
Table 7.1.

Apart from the extra memory required, the kernel is transparent to the application

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 7.1 Runtime kernel size and processor breakpoint support

program if processes on different processors communicate with each other in the 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 from the configurer, otherwise breakpoint debugging may not be used.

7.6.2   Hardware breakpoint support

Certain transputers have built-in instructions for breakpointing (see Table 7.1). 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 encountering 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.

7.6.3   Compiling the program

Programs to be debugged using breakpointing must *not* be compiled or linked using the 'Y' option. The compiler default is to create code with full debug data,

including interactive support.

All modules in a program must be compiled in the same or a compatible error mode. Error modes are checked at link time and incompatible modes prevent the link completing successfully.

7.6.4   Configuring the program

Programs to be debugged using breakpoint debugging must *not* be configured using the 'Y' option.

7.6.5   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.

7.6.6   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 *iq* 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 14.3.6.

7.6.7   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		Monitor page commands	
TOGGLE BREAK	Set/clear breakpoint.	B	Breakpoint menu.
RESUME	Execute from breakpoint.	J	Execute program.
CONTINUE FROM	Execute from current line.	S	Show debug messages.
MODIFY	Modify variable.	U	Update register display.
		W	Write to memory.

7.6.8   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 program error, see section 7.11) the process pauses and the source code may be displayed.

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

Note: A side effect of pausing at a breakpoint or error is that the debugger suspends `iserver` communications in order to preserve debugger output to the screen.

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.

7.7   Program termination

Program termination is signalled to the debugger by the termination of `iserver` (explicitly in the user code). 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.

7.8   Symbolic facilities

Symbolic debugging is debugging at source code level using the symbols de-
fined 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 spe-
cific keyboard function keys (e.g. INSPECT). Keyboard layouts for specific ter-
minal 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.

7.8.1   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 determines 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.

Addresses of important segments of code can be determined using the Moni-
tor 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 re-
locating 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 7.10.

7.8.2   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.

7.8.3   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 waiting at the end of the channel. Symbols to be inspected must be in scope with the source line last located to.

If the debugger is used in breakpoint mode variables may also be modified.

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'.

7.8.4   Tracing procedure calls

Two functions assist in the tracing of procedure or 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 |. |

7.8.5   Modifying variables

The | MODIFY | function allows variables to be changed in transputer memory and the program continued with the new values.

7.8.6   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.

7.9   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.

7.9.1   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. This is indicated by the presence of an asterisk.

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 7.2.

```
Toolset Debugger : V2.02.00      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 7.2 Example post-mortem Monitor page display for a T800 processor

Items displayed on the startup page and their meanings are summarised in Table 7.2. 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.

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 7.2 Items displayed at the Monitor page

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. 'NotProcess' `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 word '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 14.4.

- If **Wdesc** contains the word '**NotProcess**' it means that there were no runnable processes at that instant on the transputer (check timer and external links for any waiting processes). See 7.10.2.

- If **WdescIntSave** contains the word '**NotProcess**' it means that a low priority process was not interrupted when the high priority process started running.

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'.

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, (see table 7.1). 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 (Workspace)
> Program code
> Vectorspace
> Static area
> Heap area
> Configuration code
> Freespace

7.9.2 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 14.

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 7.10.

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.
←	Go to the next lowest numbered processor.
→	Go to the next highest numbered 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.

Changing to post-mortem debugging

When a program crashes during interactive debugging you are able to change to post-mortem debugging using the following command:

| Y | Postmortem debug current breakpoint session. |

7.10 A method for debugging halted programs

7.10.1 Inspecting other processes

Most transputer programs consist of several processes running in parallel, either on the same transputer or on a multitransputer network. The debugger only gives access to one process at a time; in order to inspect variables in other processes the debugger must be 'located to' the process.

For systematic debugging it can be useful to locate all processes in the network in turn and determine their status.

7.10.2 Locating 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.

Waiting for communication on a channel

Processes waiting for a internal communication can be located from source level
using the $\boxed{\text{CHANNEL}}$. 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 forget-
ting all those processes waiting on all the internal channels, then any processes
still unaccounted for must either have finished or failed to start. These remain-
ing processes cannot be located to because there are no **Wdescs** for them, and
they must be accounted for by a process of elimination.

7.10.3 Locating to procedures and functions

When a procedure is called, the workspace pointer is moved. If the debugger
locates inside a procedure or 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 procedure or function use
the $\boxed{\text{BACKTRACE}}$ key to locate to a position where the desired variable or channel
is in scope. To relocate back into the procedure or function use the $\boxed{\text{RETRACE}}$
key.

7.11   Library functions

Four procedures are provided in the occam library to assist with debugging.

DEBUG.STOP and DEBUG.ASSERT are used to stop a process, the latter on the failure to meet a specified condition; such events are treated as a program error by the debugger. DEBUG.MESSAGE is used to insert debugging messages and DEBUG.TIMER is used to aid debugging deadlocked programs. The procedures are accessed by incorporating the directive #USE "debug.lib".

Function	Description
DEBUG.ASSERT	Stops the process and alerts the debugger if the parameter evaluates FALSE.
DEBUG.STOP	Stops the process and alerts the debugger.
DEBUG.MESSAGE	Inserts debugging messages in the program.
DEBUG.TIMER	Places process on timer queue.

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 condition parameter evaluates to FALSE.

The following short example illustrates their use. (An example illustrating the use of DEBUG.TIMER is given in section 7.17.5).

```
------------------------------
--
--  Debugger example:  debug.occ
--
--  Example of debug support procedures when used with
--  and without the debugger.
--
------------------------------


#INCLUDE  "hostio.inc"
#USE      "hostio.lib"
#USE      "debug.lib"


PROC debug.entry (CHAN OF SP fs, ts, []INT free.memory)
  BOOL x :
  SEQ
    --  FALSE will cause DEBUG.ASSERT to fail assertion test
    x := FALSE
```

```
so.write.string.nl (fs, ts, "Program started")

DEBUG.MESSAGE ("A debug message only within the debugger")

so.write.string.nl (fs, ts,
                    "Program being halted by DEBUG.ASSERT ()")
DEBUG.ASSERT (x)

so.write.string.nl (fs, ts,
                    "Program being halted by DEBUG.STOP ()")
DEBUG.STOP ()

so.exit (fs, ts, sps.success)
:
```

In this example if **x** is TRUE **DEBUG.ASSERT** evaluates to TRUE and the program runs until it encounters **DEBUG.STOP**. If **x** is FALSE (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 procedure 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.

Details of the procedures can be found in part 2, section 1.10.

7.11.1 Action when the debugger is not available

If the debugger is not available on the system the debug library procedures 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.
DEBUG.TIMER	Places process on timer queue.

7.12 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 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.

7.12.1 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 23.

7.12.2 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.

7.12.3 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 transputer instruction at a time using the 'S' command.

7.12.4 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).

7.13 Debugging using embedded messages

This section describes an approach to debugging occam programs for use in those situations where breakpoint debugging cannot be used.

Programs can be debugged using messages inserted at strategic points in the program. These messages are output when the program runs and help to determine changes in the program's activity, such as the assignment of variables and the calling of procedures.

This method is easily applied to programs running on single transputers and connected directly to the host, but is less easy to use with programs running on transputer networks. In transputer networks only the root transputer communicates directly with the host, and messages from distant processes must be passed back to the root transputer through the intervening network.

A programming solution to the problem in occam is to pass the messages to a process that stores them for later retrieval. The process can be run on each transputer in the network that is to be debugged and could use a circular buffer to optimise storage and record only the recent activity of the program.

The program could be coded as two processes; one that stores messages coming from each transputer (the 'buffer manager' process), and another that formats messages for presentation to the debugger. The 'buffer manager' process would

run on each transputer running a debuggable process, whereas the message formatter would run centrally and service all transputers in the network.

7.13.1 Reading the message buffers

For programs that fail and set the error flag the debugger can read the message buffers by locating to the code that produced the error. For programs that terminate normally, the buffers can be located using the debugger Monitor page command 'L' to locate to a process pending on the host link. The buffer manager process can then be brought into scope, the message buffer located in memory and dumped to a file for reading.

7.14   Debugging example

This example illustrates some of the post-mortem and breakpoint features of the debugger. The debugger is invoked in breakpoint mode.

7.14.1 The example program

The example program 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.

Note: The example is intended for running on a B008 board wired *subs*. See section 14.4 if your system is different.

The program incorporates five processes, each coded as a separate PROC. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result. The program is listed below.

Note: Triple braces in the listing indicate fold marks in the program. They are retained for compatibility with the folding editors often used for writing occam programs.

```
-------------------------------
--
--  Debugger example:  facs.occ
--
--  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 IO
--    |                                             |
--    -----------------------------------------
--
-------------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"


PROC facs.entry (CHAN OF SP fs, ts, []INT free.memory)

  VAL stop.real    IS -1.0 (REAL64) :
  VAL stop.integer IS -1 :

  --{{{  FUNC factorial  -  compute factorial
  REAL64 FUNCTION factorial (VAL INT n)
    REAL64 result :
    VALOF
      SEQ
        result := 1.0 (REAL64)
        SEQ i = 1 FOR n
          result := result * (REAL64 ROUND i)
      RESULT result
  :
  --}}}
  --{{{  PROC feed       -  source stream of integers
  PROC feed (CHAN OF INT in, out)
    INT n :
    SEQ
      in ? n
      SEQ i = 0 FOR n
        out ! i
      out ! stop.integer
  :
  --}}}
  --{{{  PROC facs       -  generate stream of factorials
  PROC facs (CHAN OF INT in, CHAN OF REAL64 out)
    INT x :
    REAL64 fac :
    SEQ
      in ? x
      WHILE x <> stop.integer
        SEQ
          fac := factorial (x)
```

```
              out ! fac
              in ? x
        out ! stop.real
  :
  --}}}
  --{{{  PROC square      -  generate stream of squares
  PROC square (CHAN OF REAL64 in, out)
    REAL64 x, sq :
    SEQ
      in ? x
      WHILE x <> stop.real
        SEQ
          sq := x * x
          out ! sq
          in ? x
      out ! stop.real
  :
  --}}}
  --{{{  PROC sum         -  sum input
  PROC sum (CHAN OF REAL64 in, out)
    REAL64 total, x :
    SEQ
      total := 0.0 (REAL64)
      in ? x
      WHILE x <> stop.real
        SEQ
          total := total + x
          in ? x
      out ! total
  :
  --}}}
  --{{{  PROC control     -  user interface and control
  PROC control (CHAN OF SP fs, ts,
                CHAN OF REAL64 result.in,
                CHAN OF INT n.out)
    REAL64 value :
    INT n :
    BOOL error :
    SEQ
      so.write.string.nl (fs, ts,
                "Sum of the first n squares of factorials")

      error := TRUE
      WHILE error
        SEQ
          so.write.string (fs, ts, "Please type n: ")
          so.read.echo.int (fs, ts, n, error)
          so.write.nl (fs, ts)

      so.write.string (fs, ts, "Calculating factorials ...")

      n.out ! n
      result.in ? value

      so.write.nl (fs, ts)
```

```
so.write.string (fs, ts, "The result was: ")
so.write.real64 (fs, ts, value, 0, 0)    --  free format
so.write.nl (fs, ts)
so.exit (fs, ts, sps.success)
:
--}}}

CHAN OF REAL64 facs.to.square, square.to.sum, sum.to.control :
CHAN OF INT feed.to.facs, control.to.feed :

PAR
  feed (control.to.feed, feed.to.facs)
  facs (feed.to.facs, facs.to.square)
  square (facs.to.square, square.to.sum)
  sum (square.to.sum, sum.to.control)
  control (fs, ts, sum.to.control, control.to.feed)
:
```

7.14.2 Compiling the `facs` program

The source of the program is provided on the toolset examples directory. 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.

Using `imakef`

If your system has a MAKE utility you may use `imakef` to generate a suitable Makefile to help build the program:

```
imakef facs.bah

make -f facs.mak                              (UNIX)
make /f facs.mak                              (MS-DOS/VMS)
```

Using the tools directly

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 does provide useful information on the progress of the linking operation.

Command sequences follow for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

UNIX:

```
oc -ta facs.occ -o facs.tah
ilink -ta facs.tah hostio.lib convert.lib -f occama.lnk
     -o facs.cah
icollect -t facs.cah -o facs.bah
```

MS-DOS/VMS:

```
oc /ta facs.occ /o facs.tah
ilink /ta facs.tah hostio.lib convert.lib /f occama.lnk
     /o facs.cah
icollect /t facs.cah /o facs.bah
```

7.15   Breakpoint debugging

The following section demonstrates how to debug the example **facs** program in breakpoint mode. This example of breakpoint debugging assumes the hardware configuration shown in figure 7.3.
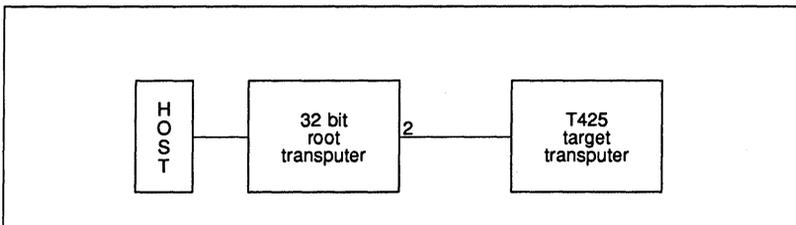
Figure 7.3 Hardware configuration for breakpoint example

7.15.1   Prerequisites for breakpoint debugging

You should ensure that the appropriate environment variables described in section 2.10.4 have been initialised before you proceed.

7.15.2   Loading the program

The program is loaded for breakpoint debugging by invoking **idebug** with the Breakpoint option in the commands given below. Use the appropriate command for your system.

```
        idebug -sr -si -b2 facs.bah -c t425        (UNIX)
        idebug /sr /si /b2 facs.bah /c t425     (MS-DOS/VMS)
```

The 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 14.3 for more details about the options to use if in doubt.

7.15.3 Setting initial breakpoints

Initial breakpoints can often be set by invoking the Monitor page '`B`' command and specifying an entrypoint breakpoint (this would set a breakpoint at `facs.entry`). In this example a different method is used 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.tah`. The debugger uses debug information within the object module to select the source file.

The source file `facs.occ` is displayed with the cursor positioned at the first procedure definition, namely `facs.entry`. At this point the program is still waiting to be started.

Use GOTO LINE to move the cursor to line 56 (`out ! fac`) and set a breakpoint there using TOGGLE BREAK. The debugger confirms the breakpoint is set.

7.15.4 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.

7.15.5 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.

7.15.6   Inspecting variables

Variables and channels in **facs** can now be examined. For example, to examine the variable **fac** move the cursor to **fac** and press INSPECT. The debugger displays the value as **REAL64 1.0** and gives its address. Pressing INSPECT with the cursor positioned on a space causes the debugger to prompt you for a symbol.

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 line 56 in the procedure **facs**.

7.15.7   Backtracing

facs is called in parallel by **facs.entry** 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.entry** where **facs** is called. Press TOP to return to where the breakpoint occurred. The current location point is line 56 in the procedure **facs**.

7.15.8   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**. 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**. The debugger jumps down the channel connecting the two processes and locates to the corresponding channel input in procedure **square** (**in ? x** statement). Variables in scope within **square** now become available for inspection (at this stage they have not been initialised).

7.15.9   Modifying a variable

In breakpoint debugging program variables may be modified. Start by first inspecting **x** in order to ensure that the new value will be different. To modify the variable **x** position the cursor on **x** and press MODIFY. At the modify value prompt specify the value to be placed in **x**. Note that the modify prompt reminds you of the type of **x**. Give any valid value and check the value has changed by inspecting **x** once again.

7.15.10 Entering #INCLUDE files

Press GOTO LINE and select line 17. This will locate you to the
#INCLUDE "hostio.inc" line. By using the ENTER FILE key you may now
enter the #INCLUDE file (and any nested files within it if they were present);
the EXIT FILE key will bring you out again into the enclosing file.

7.15.11 Resuming the program

To resume execution of the program from the current breakpoint press
the RESUME key. This will cause the program to resume until it encounters
the breakpoint again. Press an appropriate key to enter the symbolic debugging
environment. This will cause the debugger to locate to line 56.

7.15.12 Clearing a breakpoint

To clear the breakpoint already set at line 56 use the TOGGLE BREAK key. The
debugger will confirm that the breakpoint has been cleared. Press RESUME to
resume execution and cause the program to display its result.

The debugger will confirm that the program has finished and will pause in order
to enable you to read the output from the program. Press any key as indicated
to enter the Monitor page. Note that the Monitor page displays the exit status
from the program.

7.15.13 Quitting the debugger

Finally, to quit the debugger you can use the Monitor page 'Q' command. You
may also quit the debugger from Symbolic mode by using the FINISH key.

7.16   Post-mortem debugging

The following section demonstrates how to debug the example `facs` program in post-mortem mode. This example of post-mortem debugging assumes the hardware configuration shown in figure 7.4.
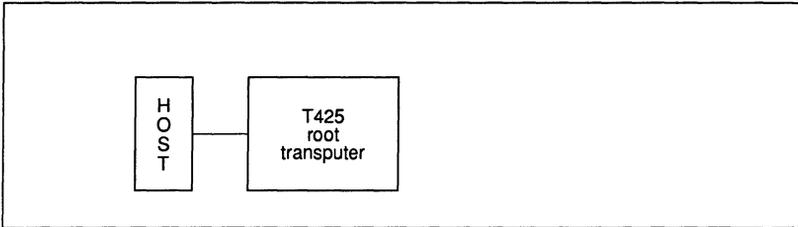
Figure 7.4 Hardware configuration for post-mortem example

7.16.1   Prerequisites for post-mortem debugging

You should ensure that the appropriate environment variables described in section 2.10.4 have been initialised before you proceed.

7.16.2   Running the example program

When you have built an executable code file you can run the program by typing one of the following commands:

```
iserver -se -sb facs.bah                         (UNIX)
iserver /se /sb facs.bah                    (MS-DOS/VMS)
```

The program immediately prompts you for a value. For correct execution the number must be less than 100.

To create an error for the purpose of this tutorial, give the value 101 and press RETURN. The program will fail with the message:

```
Error - iserver - Error flag raised by transputer.
```

7.16.3   Creating a memory dump file

To create a memory dump file for the debugger to read, type:

```
idump facs 15000
```

This creates a file called `facs.dmp` containing the transputer's register contents

and the first 15000 bytes of memory. You are then returned to the operating system prompt.

7.16.4 Running the debugger

To debug the example program, use one of the following commands:

```
idebug -si facs.bah -r facs -c t425          (UNIX)
idebug /si facs.bah /r facs /c t425        (MS-DOS/VMS)
```

The **iserver** 'si' switch is optional. The 'r' option identifies the program as one that was executed on the *root* transputer and specifies the memory dump file to be read.

Note: If your transputer is not a T425 you should change the **T425** option to the appropriate transputer type.

Should you wish to invoke the debugger a second time on this single processor example, without an intervening **idump** command, you will need to add the **iserver** 'sr' option to the command line (see section 14.3.5).

The debugger first displays its version number, then some processing information, and eventually locates to the source line from which the error was generated:

```
sq := x * x
```

You can now begin to debug the program. You can use the symbolic facilities to browse the source, locate to specific lines and areas of code, inspect variables and channels, and trace procedure calls, and you can inspect and disassemble memory using the Monitor page commands.

The following sections illustrate some of the debugging operations you can perform on the example program. For further details about any of the debugging functions described in these sections, see chapter 14.

Inspecting variables

When the debugger is displaying source code, you may inspect any variable by placing the cursor on the variable and pressing INSPECT.

For example, to display the value of **x**, place the cursor over **x** in the source code and press INSPECT. **x** is displayed in both decimal and hexadecimal forms, and

its address in memory is given in hexadecimal. For example:

```
REAL64 'x' has value ...
9.3326215443944096E+155 (#605166C698CF1838) (at
#80000464)
```

In the same way you can inspect the values of **sq**, **square**, **stop.integer**, **stop.real**, and any other variable that is in scope. Use the cursor keys to scroll through the code. To return to the source of the original error, use the RELOCATE function.

You can also use the INSPECT function to examine procedures and functions. If you place the cursor on a procedure or function name and press INSPECT, the debugger displays its address and workspace requirements.

You can also examine any symbol in the source by specifying its name. To do this, move the cursor to a blank area and press INSPECT. The debugger then prompts for the symbol name.

Inspecting channels

The debugger can also examine processes on channels within the scope of the original error. If you place the cursor on channel **out** and press INSPECT, information about the channel is displayed. For example:

```
CHAN 'out' has Iptr:#800022F8 and Wdesc:#80000381
    (Lo) (at #8000063C)
```

This indicates that there is a process waiting for communication on channel **out**, and that it is a low priority process. To find out which occam process is waiting, press CHANNEL. The cursor will be placed on the line corresponding to the other process, which in this example is inside the procedure **sum**, on the following line:

```
in ?  x
```

Within procedure **sum**, you can examine any symbol using INSPECT.

Within the **sum** procedure you can inspect the channel **out** and use CHANNEL to jump to the waiting process, which is the procedure **control** that is waiting for the final result. Again you can use INSPECT to examine any symbol.

Retracing and Backtracing

So far the debugger has located three of the five processes that compose the program. What about the others?

First use the RETRACE key to retrace your steps back to procedure **square**.

When in procedure `square`, inspect channel `in`, which is connected to the `facs` procedure. It is empty, which means that no process is waiting to communicate.

Next try BACKTRACE. This function backtraces down nested procedure calls. Each time the function is invoked the cursor is placed on the line in the enclosing code from which the procedure was called.

In this example, BACKTRACE moves the cursor to the line where procedure `square` is called. Again, you can inspect any symbol which is in scope at this line. For example, you can inspect the channels `feed.to.facs` and `facs.to.square`. Both should be empty, which means that the remaining processes were actively executing, rather than waiting to communicate, when the program halted.

To find the active processes, you need to examine the transputer's process queues using the Monitor page facilities, as described below.

Displaying process queues

To display the process queues, first enter the debugger Monitor page from the symbolic environment by pressing the MONITOR key. Low level information is displayed for the current processor, along with a list of Monitor page commands.

To display the processor's active process queues, use the Monitor page '`R`' command. This displays two active processes, identified by their respective `Iptr` and `Wdesc`. When you have identified the processes to examine, you can use the Monitor page '`G`' command to jump to those processes and inspect the code.

Other commands to try from the Monitor page are '`T`', which displays the processes waiting on the transputer's timers; and '`L`', which displays processes waiting for communication on the transputer's links.

Goto process

When you press '`G`', the following message is displayed:

```
[CURSOR] then [RETURN], or 0 to F, (I)ptr, (L)o,
or (Q)uit
```

To jump to a specific process and display the source code associated with that process, place the cursor on an `Iptr` and press RETURN.

Commands '`I`' and '`L`', allow you to jump to the main process or low priority process respectively, and commands '`0`' – '`F`' allow you to display specific lines on the right hand side of the display.

To display the first active process, type '0' (zero). The cursor will be placed on the following source line (in procedure 'feed'):

```
out ! i
```

Because this process is on the queue and not waiting, it must have already performed the communication and is about to resume executing. You can examine variables within the procedure as before.

To display the last remaining process in the program, press MONITOR again, and type 'G' followed by '1' to locate to the second process in the queue.

This process will either be executing code within the compiler libraries or within the replicated SEQ. If it is executing code within a library, the debugger displays the call to the library routine rather than the source itself, because the source is not supplied. For example:

```
result := result * (REAL64 ROUND i)
```

Again, you may inspect variables within the process. For example, by inspecting the variable 'i', you can determine how many times the loop has been executed. Or you can use BACKTRACE to determine where the function was called from.

Other symbolic functions

Other symbolic functions that you may like to try while you are in the debugger are listed below.

TOP	Returns to the error location, or last location selected by Monitor page 'G' command.
INFO	Displays Iptr, Wdesc, and priority, of the last position located to, together with the processor type and number.
SEARCH	Allows you to search forward through the file for a specific string.
HELP	Displays a summary of debugger function keys.
GET ADDRESS	Displays the memory address of the transputer code corresponding to the current source line.
CHANGE FILE	Allows you to examine any file.
ENTER FILE	Allows you to open and examine included files.

EXIT FILE	Allows you to close included files.
GOTO LINE	Moves to a particular line of the file.
TOP OF FILE	Moves to the first line of the file.
BOTTOM OF FILE	Moves to the last line in the file.

7.17 Hints and further guidance

This section contains some hints about specific debugging operations and some guidelines to follow when analysing deadlocks in occam programs.

7.17.1 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.

7.17.2 Examining and disassembling memory

Within the Monitor page environment, the debugger keeps a record of two memory addresses; the start address of the last disassembly, used as the default by the 'D' command, and the address of the last region of memory to be displayed, used by the 'A', 'H', and 'I' commands.

This allows you to switch easily between code disassembly and memory display. You can, for example, disassemble a portion of memory using the 'D' command, examine its workspace in hex using the 'H' command, and then return to the original address by invoking the 'D' command once again.

7.17.3 occam scope rules

The debugger can only display the values of variables that are in scope. For example, division by zero in the following procedure r would cause an error, and the debugger would locate to that source line.

```
--------------------------------
--
--   Debugger example:  scope.occ
--
--   Example of occam variable scope rules.
--
--------------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"


PROC scope.entry (CHAN OF SP fs, ts, []INT free.memory)

  PROC p ()
    INT a :

    PROC q (VAL INT b)
      INT c :
      SEQ
        c := b + a
    :

    PROC r (VAL INT d)
      INT e :
      SEQ
        e := 0
        e := d / e    --  <== The debugger will locate
    :                 --      to here after the error

    INT x :
    SEQ
      x, a := 99, 57
      INT y :
      SEQ
        y := 42
        q (y)
      r (x)           --  <== And backtrace to here
  :

  SEQ
    p ()
    so.exit (fs, ts, sps.success)
:
```

At the line that contains the division by zero, variables e, d, and a are in scope
and may be inspected, but variables x, y, c, and b are out of scope and cannot
be inspected.

If the debugger now located to the *call* of *r*, the only variables in scope and accessible for inspection would be **a** and **x**.

7.17.4 Debugging IF and CASE statements

IF constructs with no **TRUE** guards, and **CASE** constructs where no selections are matched, stop the program as though a **STOP** statement had been encountered in the program. This avoids the need to create a default case each time the statements are used.

However, it can be useful for the purpose of debugging these statements, to use a default case. If a default is specified, the debugger can locate directly to the **STOP** statement within the construct, which indicates exactly where the error occurred.

7.17.5 Analysing deadlock

Deadlocks that occur in multitransputer networks can be debugged by using the Monitor page 'L' command to examine processes on the transputer links. Deadlocks in single transputer programs are more difficult to debug because there is no way to enter the program; there are no active processes from which to inspect channels, and no links to other transputers to provide an alternative entry point.

In practice, it is often obvious to the programmer which channel or channels are causing deadlock, and a dummy process can be added to the program to provide an entry point for the debugger.

Consider the following code:

```
-------------------------------
--
--  Debugger example:  deadlock.occ
--
--  Example of deadlock.
--
-------------------------------


#INCLUDE "hostio.inc"
#USE     "hostio.lib"


PROC deadlock.entry (CHAN OF SP fs, ts, []INT free.memory)

  PROC deadlock ()
    CHAN OF INT c :
    PAR
      SEQ
        c ! 99
        c ! 101

      INT x :
      SEQ
        c ? x
    :                 --  <==  Missing second input

  SEQ
    deadlock ()
    so.exit (fs, ts, sps.success)
  :
```

The program can be debugged by adding a process that will remain idle (by waiting on a TIMER) while the program is debugged. An example of the type of code that is required is illustrated below.

```
-----------------------------
--
--   Debugger example:   deadfix.occ
--
--   Example of deadlock and how to provide
--   debugging support.
--
-----------------------------


#INCLUDE  "hostio.inc"
#USE      "hostio.lib"
#USE      "debug.lib"


PROC deadfix.entry (CHAN OF SP fs, ts, []INT free.memory)

  PROC deadlock.debug ()
    CHAN OF INT c :
    CHAN OF INT stopper :
    PAR
      DEBUG.TIMER (stopper)      --  Hook for debugger
      SEQ
        PAR
          SEQ
            c ! 99
            c ! 101

          INT x :
          SEQ
            c ? x
            --  <== Missing second input

        stopper ! 0  --  terminate debug.timer
  :

  SEQ
    deadlock.debug ()
    so.exit (fs, ts, sps.success)
  :
```

The procedure DEBUG.TIMER is supplied in the occam library debug.lib.
The process lies dormant on the processor's timer queue waiting for a time as
far into the future as the processor can provide. When the timeout expires, the
process places itself back on the timer queue.

Such a process provides a hook into the program for locating deadlocked pro-
cesses because the process is always accessible to the debugger on the timer

queue. By locating to it you can access variables which are in scope at the point of its execution and thereby detect the deadlock. In the modified program a deadlock still forms in the procedure, but there is now a way to enter the program.

To enter the program and inspect the deadlock, first invoke the Monitor page environment, and use the Monitor page 'T' command to inspect the transputer's timer queue, on which there will be a process waiting. Use the 'G' command to go to that waiting process, and the debugger will locate to the call of **DEBUG.TIMER**.

You can then use INSPECT to examine the channel **c** where the program has deadlocked, and which will therefore contain the process that is waiting for communication. Finally you can use CHANNEL to jump to the deadlocked process.

The compiler does not insert this kind of debugging code automatically, for several reasons. Firstly, it is the philosophy of the occam toolset not to alter the runtime code in any way. Secondly, most programs use many channels, and the execution overheads and code size could become unacceptably large. Again for the above example code this would be unimportant because the process consumes no CPU time, but this may not be true for many programs. Lastly, it could be difficult to distinguish the true deadlocked process from the many idle debug processes waiting on the timer queues.

7.17.6   Inspecting soft configuration channels

Soft channels declared at the configuration level (i.e. those internal to a processor which are not placed on its external links) may be inspected from the Monitor page by knowing that they are located near the beginning of the *Configuration code* area which appears after the user *Program code* area (as displayed by the Monitor page Memory map command).

7.18   Points to note when using the debugger

This section contains some extra information which may be of use when using the debugger.

7.18.1   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 synchro-

nised.

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.

7.18.2   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.

7.18.3   Using INSPECT with channel communications

When debugging a program compiled for interactive debugging it should be remembered that any channel communication is achieved via library calls. As a consequence, the INSPECT key may display an **Iptr** relating to code in the debugging kernel system rather than the **Iptr** of a user process waiting on the channel. This may lead to channel communications not involved with an **ALT** appearing to having the same process **Iptr** (the **Wdesc** will be valid and unique). In order to correctly establish the **Iptr** of the process waiting at the other end, you should use the CHANNEL key to locate to the process followed by the INFO key to obtain process details.

7.18.4   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.

7.18.5 Minimal confidence check

A first level confidence check to perform with a program which is misbehaving is to perform a compare memory check using the Monitor page 'C' command. This will help to highlight any memory corruption problems which may occur due to faulty memory or faulty program logic. You should always ensure that no compiler checks have been disabled to prevent the latter.

7.18.6 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.

7.18.7 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 ('Y') 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 14.3.6 for more details).

### 7.18.8	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. In this situation the debugger fails to update the screen other than to put the following message at the top of the screen when it attempts to display the Monitor Page:

```
Toolset Debugger :  V2.02.00 Processor n "name" (Tm)
```

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.

### 7.18.9	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.

### 7.18.10	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.

### 7.18.11	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.

7.18.12 Phantom breakpoints

Because of the mechanism used for breakpoints on those transputers without hardware breakpoint support (see Table 7.1) 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 oc and other TCOFF compatible INMOS compilers such as icc.

The following occam code generates a phantom breakpoint.

```
WHILE TRUE
  SKIP
```

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.

7.18.13 Breakpoint configuration considerations

When breakpoint debugging you should remember that the root transputer of a network is used by the debugger for its own purposes.

On some transputer motherboards with an inbuilt pipeline, the root transputer is normally booted down link 0; subsequent transputers in the pipeline boot down link 1. This may (accidently) be a problem if you simply take a network configuration which was not configured with breakpoint debugging in mind (eg. a pipeline configuration) and attempt to breakpoint debug it. The debugger will in effect, attempt to skip load it onto the rest of the network; the program may load (if by chance the right link connections are available), if the boot link is different

it will not be able to talk to the host (iserver) when it executes.

Such an event may easily be verified by using the Monitor page 'L' option when positioned on processor 0. This will indicate whether the root transputer booted from a different link to that specified in the configuration file.

When breakpoint debugging, the debugger will warn you if the boot is different from that expected for the root processor, before the network is loaded.

7.18.14 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 `icollect` 'P' option.

Alternatively you may use the debugger command line dummy debug 'D' option.

7.18.15 Long source code lines

Source code lines longer than 500 characters cause the symbolic source code browser to treat them as multiple lines and subsequently it will loose line synchronisation; (i.e. it displays incorrect line number information).

7.18.16 Setting breakpoints on the transputer `seterr` instruction

The debugging kernel does not resume the transputer `seterr` instruction with its original (correct) `Iptr` (it resumes it with an `Iptr` within the kernel area). Because a kernel operates in Halt-on-Error mode, the `seterr` instruction has the effect of halting the processor. The effect of the incorrect `Iptr` is only apparent if you subsequently switch to post-mortem debugging whereupon the debugger will complain that it is unable to locate to an `Iptr` within the kernel area. If this is a problem, you should note the `Iptr` before resuming from the breakpoint.

This problem will occur if you resume from a breakpoint on an occam STOP statement which has been compiled in either Halt or Universal error modes.

7.18.17 Backtracing to occam configuration code

When used in conjunction with the occam configurer `occonf`, `idebug` is able to backtrace to the occam configuration source code. Symbolic debug information provided by the configurer at this level is, however, not complete (although sufficient to enable correct source line locating), and as a consequence you are unlikely to be able to inspect variables, channels or constants.

8 Access to host services

This chapter describes how programs communicate with the host computer via the host file server and the i/o libraries. It briefly describes the protocols used, outlines how to place host channels on a transputer board, and discusses how processes can be multiplexed to a single host.

8.1 Introduction

occam, like most high level programming languages, is independent of the host operating system. At the programming level, communication with the host is achieved via a set of i/o libraries that are provided with the toolset. The libraries in turn use the services provided by the host file server.

The host file server and the functions it provides are transparent to the programmer. The server functions are activated whenever a program is loaded using the **iserver** tool. Programs that use the i/o libraries should always be loaded using **iserver**.

For an example of a program that communicates in a simple way with the host computer, including details of how it is compiled, linked and loaded, see chapter 4.

8.2 Communicating with the host

Programs communicate with the host through i/o library routines that in turn use functions provided by the host file server.

8.2.1 The host file server

The host file server provides the runtime environment that enables application programs to communicate with the host. It contains functions for:

- Opening and closing files

- Reading and writing to files and the terminal

- Deleting and renaming files

- Returning information from the host environment, such as the date and time of day

- Returning information specific to the server, such as a version number

- Starting and stopping the server.

Details of the server functions can be found in part 2 appendix H.

8.2.2   Library support

Two i/o libraries are provided for accessing the file system and other host services. The libraries are summarised below.

> `hostio.lib`   File and terminal i/o; host access
>
> `streamio.lib`   Stream-based terminal and file i/o

All routines in these libraries are independent of the host operating system.

The hostio library contains basic routines for accessing files and controlling the file system. It also contains routines for general interaction with the host. Use the hostio library for basic file operations, and for accessing host services.

The streamio library contains routines for creating and outputting to streams. It also provides primitives for reading and writing text and numbers, and for controlling the screen. Use the streamio library for inputting and outputting character and data streams.

Definitions of constants and protocols used within the libraries are provided in the include files `hostio.inc` and `streamio.inc`. These files should be included in all programs where the respective libraries are used.

Details of all i/o procedures and functions can be found in part 2 chapter 1.

8.2.3   File streams

The host file server supports a stream model of file and terminal access. When a file is opened a 32-bit integer stream id is returned to the program. This identifier must be quoted by the program whenever the file is accessed, and is valid until the file is closed.

Streams and files must be explicitly closed by the programs that use them, and the server must be explicitly terminated when the program finishes and host

services are no longer required.

Three streams are predefined:

spid.stdin   standard input

spid.stdout   standard output

spid.stderr   standard error

These streams can be closed by the programmer, but cannot be reopened. Take care not to close the standard streams if you are using hostio routines that read or write to them. The streams can only be closed by specifying the streamid explicitly and cannot be closed inadvertently using the hostio routines.

Standard input and output are normally connected to the keyboard and screen respectively, but may be redirected by the operating system.

Streams and files other than the three standard streams described above must be explicitly closed by the program. When the program finishes and host services are no longer required, the server should be terminated by the transputer application calling **so.exit**.

Protocols

occam programs communicate with the host file server through a pair of occam channels. Requests for service are sent to the host on one channel and replies are received on the other. Both channels use the **SP** protocol, which is defined in the include file **hostio.inc**.

8.3   Host implementation differences

The IBM PC version of the host file server supports a number of DOS specific commands. For details of the routines provided for this implementation see the Delivery Manual that accompanies the release. The VAX VMS and Sun 3 UNIX implementations have no host specific commands.

If you wish to write programs that are portable between all implementations of the toolset you are recommended to use only host independent routines. All procedures and functions in the hostio and streamio libraries are host independent.

8.4   Accessing the host from a program

For programs to be run on transputer boards the host is accessed through the channels **fs** and **ts**, both defined as **CHAN OF SP**. Protocol **SP** is defined in the include file **hostio.inc**.

For single transputer programs the channels are defined within the program, and for multiprocessor programs the channels are placed on the link that is connected to the host. The normal location for the connection to the host is link zero on the root processor.

8.4.1   Using the simulator

The simulator tool **isim** provides access to the host file server in the same way as a single processor program running on a board, with the following channel placements: **fs** at **link0.in**; **ts** at **link0.out**.

8.5   Multiplexing processes to the host

The host file server is a single resource, connected to a process running on the root transputer via a pair of occam channels. This is illustrated in figure 8.1.
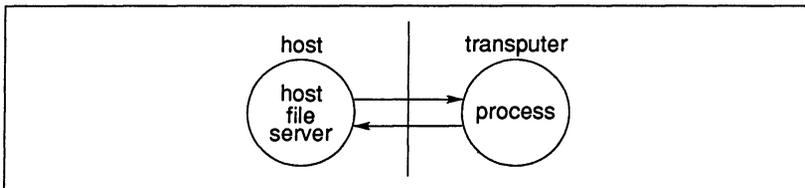
Figure 8.1 Program input/output

If more than one process requires access to the host then the server must be shared between a number of processes, ensuring that all processes are served in turn. The simplest solution where a resource is used by more than one process is to provide a multiplexor.

A multiplexor is a process which takes many inputs and connects them to a single shared resource and ensures that communications from different processes do not conflict.

Two routines that allow multiple processes to communicate with the host via the host file server channels are provided in the hostio library. The routines are called **so.multiplexor** and **so.overlapped.multiplexor**. Details of the routines can be found in part 2, section 1.4.9.

An example of a multiplexed system is shown in figure 8.2 and occam code that would implement the system is listed in figure 8.3.
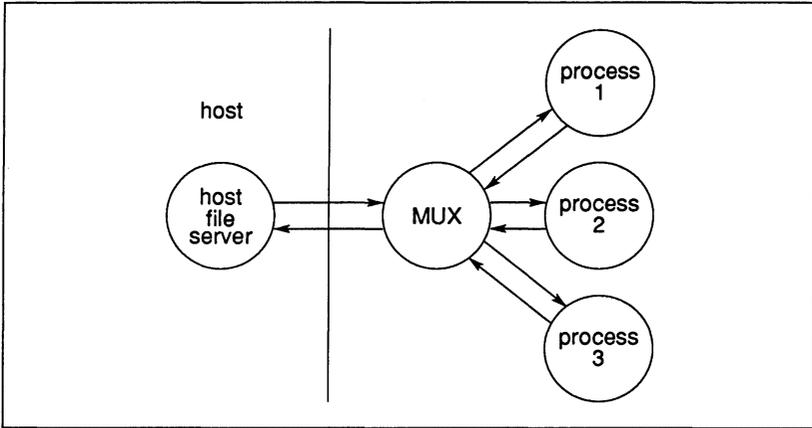
Figure 8.2 Multiplexing the host file server

Multiplexor processes can be chained together to produce any degree of multiplexing to the host. However, the host is a single, finite resource and unrestrained multiplexing of processes should be avoided if possible.

8.5.1   Buffering processes to the host

It may sometimes be useful to pass data invisibly through another process, for example when passing data to the server through intervening processes. The hostio library routine `so.buffer` takes a pair of input and output channels and passes data through unchanged.

8.5.2   Pipelining

If data has to pass through many processes before reaching the server efficiency may be improved by allowing a data transfer to begin before the previous one has completed its journey down the line of processes. This allows several data transfers to be in progress simultaneously and is known as pipelining.

The routine `so.overlapped.buffer` can pipeline several buffers up to a user-defined limit. A pipelined version of the multiplexor process called `so.overlapped.multiplexor` performs the same function for multiplexed processes.

```
#INCLUDE "hostio.inc"  -- SP protocol declaration

PROC mux.example (CHAN OF SP fs, ts,
                  []INT free.memory)

  #USE "hostio.lib"  -- host i/o libraries

  #USE "process0"    -- user processes
  #USE "process1"
  #USE "process2"

  SEQ
    CHAN OF BOOL stop:
    [3]CHAN OF SP from.process, to.process:
    PAR
      so.multiplexor(fs, ts, -- server channels
                     from.process, to.process,
                     -- multiplexed channels
                     stop)   -- termination channel

      SEQ
        PAR   -- run user processes in parallel
              -- sharing the iserver
          process0(to.process[0], from.process[0])
          process1(to.process[1], from.process[1])
          process2(to.process[2], from.process[2])
        stop ! FALSE  -- terminate multiplexor

    so.exit(fs, ts, sps.success)
:
```

Figure 8.3 Multiplexing example

9 Mixed language programming

This chapter describes how to incorporate code written in other languages into occam programs. It begins by outlining the various ways in which foreign language code can be mixed with occam, describes briefly how to mix code at the configuration level, and describes in detail the special facilities available for importing C functions (in TCOFF) into occam 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 system is desirable.

The occam programming model provides a clean and simple basis for mixing languages. The model consists of independent parallel processes, communicating via channels, which can be distributed in any way to a network of transputers. In accordance with this model, programs written in other languages can be viewed as single, separately compiled processes that can be run in parallel with each other, and with other occam processes, on any transputer in the network.

In addition special facilities and library support are provided for calling C code from occam. The system supports the calling of C functions by a compiler pragma and a set of library functions which allow static and heap areas to be initialised and their use terminated.

Code which has been generated by the TCOFF family of language compilers may be mixed at configuration level, using the ANSI C toolset configurer `icconf`. The ANSI C configuration system and language enables processes written in different languages to be placed on the same processor or on a network of processors. Further details of this configuration system is given in the ANSI C Toolset User Manual.

The toolset also provides a special set of interfaces which allow foreign language code to be incorporated into occam programs as equivalent occam processes . This enables configuration using the occam configurer `occonf`; an example is provided in the `examples` directory supplied with the toolset. The interface code system may be used to support source code written for use with earlier INMOS language compilers and toolsets. Further information on this subject is provided as an appendix to the ANSI C Toolset User Manual.

Mixed language programs can be debugged using the toolset debugger `idebug`, with some restrictions.

It is also possible to call separately compiled occam procedures from other languages. Since occam code requires no elaborate run time environment, and separately compiled procedures are re-entrant, the code for these procedures can be shared by different processes running on the same transputer.

9.2   Importing C functions

Special facilities and library support exist for calling C functions from occam. This allows existing, proven C code to be used in occam programs, with certain minor restrictions. The facilities comprise the `oc` compiler pragmas **EXTERNAL** and **TRANSLATE** and four library routines which allow C code generated by the ANSI C compiler `icc` to be incorporated. Section 9.3 lists equivalent occam data types for all C types. In certain cases no true equivalent exists and an alternative action is suggested.

Only C functions which do not require any server communication, i.e. those linked with the reduced C runtime library, can be called from occam. In addition stack checking should not be enabled on any C function to be called from occam.

Most C runtime environments automatically provide C programs and functions with a static area (for holding static data and external variables) and a heap area (for memory allocation). Since occam requires neither of these, the facilities are provided separately by four routines in the occam library `callc.lib`. The routines provide the mechanisms for setting up and terminating C static and heap areas from occam so that suites of C functions may be called. Each C function which uses static data needs to be able to find this area. In order to do this, every C function takes, as its first parameter, a pointer to the start of the static area.

Some simple C functions may not require static or heap areas and may be called more easily without using the special library routines. When calling a C function therefore, the first step is to decide whether static and heap areas are required.

9.2.1   Deciding whether a static area is required

For many C functions it may not be immediately obvious whether static or heap is required (the heap area requires a previously set static area). For example, some, but not all, library functions require static and heap areas but because it would be difficult to distinguish those that do, a static and heap area should be assumed whenever a library function is called.

Because of the difficulty in covering all types of functions, the following series of rules is offered as a way of determining whether a function requires static or heap. The rules ensures that no C function is called in the incorrect way, even though static and heap areas may be assumed when they are not actually required.

If the function uses static variables then static is required.

If the function accesses external variables static is required.

If the function uses any functions from the runtime library static and heap is required.

Functions which fail all the above tests can be assumed *not* to require static or heap, and can be called without using any of the static or heap library functions.

9.2.2 Functions which do not require static or heap

C functions which do not require static or heap can be called using the **EXTERNAL** pragma. There are two possible cases based on whether or not the function requires a global static base (gsb) pointer parameter.

Functions which are compiled normally expect gsb as the first parameter even if they make no use of static. If there is no use made of the static area, gsb will not be used but must still be passed as a dummy parameter.

9.2.3 Declaring the C function

In order for occam code to call an external function it must have a description of the function. The **EXTERNAL** pragma is provided to create descriptors for functions written in languages other than occam. The syntax of the pragma is as follows:

#PRAGMA EXTERNAL *"function declaration = workspace* [*, vectorspace*] *"*

The optional parameter **vectorspace** is not required for C functions.

For example:

```
#PRAGMA EXTERNAL "INT cfunction(VAL INT gsb, INT arg1, arg2)
= 50"
```

Note that gsb is declared as the first parameter. Workspace is a constant value and represents the number of words of workspace required by the C function (in the above example this was 50 words). Workspace must be large enough

to accommodate the workspace of any sub-calls made by the C function. The workspace requirement of a C function can be estimated as the number of words required by any automatic variables, plus an additional four words for each function call. If the C function makes a call to the runtime library, workspace should be increased by at least 150 words.

For example:

C function to be called:

```
int cfunc(int arg1, int arg2)
{
   /* body of cfunc */
}
```

occam **calling code**:

```
#PRAGMA EXTERNAL "INT cfunc(VAL INT gsb, INT arg1, arg2) =
                 50"

PROC callc()
  INT dummy.gsb, c.arg1, c.arg2, result:
  SEQ
    dummy.gsb := 0    -- dummy.gsb has a null value
    -- calculate c.arg1 and c.arg2
    result := cfunc(dummy.gsb, c.arg1, c.arg2)
    -- rest of function
:
```

It is possible to remove the need for the dummy parameter altogether by compiling the C function without a static link parameter i.e. with the **icc** compiler pragma IMS_nolink active.

For example:

C function to be called:

```
int cfunc(int arg1, int arg2);

#pragma IMS_nolink(cfunc)

int cfunc(int arg1, int arg2)
{
   /* body of cfunc */
}
```

occam **calling code**:

```
#PRAGMA EXTERNAL "INT cfunc(INT arg1, arg2) = 50"

PROC callc()
  INT c.arg1, c.arg2, result:
  SEQ
    -- calculate c.arg1 and c.arg2
    result := cfunc(c.arg1, c.arg2)
    -- rest of function
:
```

Translating C names

The **TRANSLATE** pragma can be used to convert C names to valid occam names.

Some C names may contain invalid occam characters, for example, the underscore. It is usually possible to ensure that the C function has an acceptable occam name but where this is not possible or desirable the **TRANSLATE** pragma can be used to generate an occam alias.

The syntax is as follows:

#PRAGMA TRANSLATE *occamname* "*Cname*"

For example:

#PRAGMA TRANSLATE c.func "c_func"

TRANSLATE pragmas must precede any reference to their identifier and this includes any identifier defined by an **EXTERNAL** pragma. For example:

```
#PRAGMA TRANSLATE c.routine "c_routine"
#PRAGMA EXTERNAL "PROC c.routine () = 100"
```

Linking

occam programs that call C functions must be linked with the compiled C function. For details of how to use the linker see chapter 19.

9.2.4   Functions which require static and/or heap

For C functions which require static and/or heap the space must be set up in the occam code before the function is called, and terminated when no longer required. These operations are performed by procedures supplied in the 'callc' library `callc.lib`.

The static area

C static data is stored in a reserved area of memory called the static area which must be set up by the system and initialised. Each C function then locates this static area using a pointer to its base passed by the runtime system. This pointer is called the global static base (gsb) and is implicit in C and therefore normally hidden from the user. Because C functions expect to receive this parameter it should be passed explicitly by the calling occam code. This means that a call to a C function from occam will have one extra parameter compared to an equivalent call from C. The exception to this is when the C function is compiled with the `IMS_nolink #PRAGMA`, which causes the function to be compiled without the global static base parameter.

The heap area

The heap area is that area of memory from which the C memory allocation functions reserve their memory space. It is separate from the static area and requires a static area to be previously allocated because information about the heap is held in static variables.

The heap need not be set up if it is not required, but remember that it may be used implicitly by a library call.

Callc library

The library `callc.lib` provides four occam procedures for initialising static and heap areas and terminating them after use. The routines are summarised below.

Function	Description
`init.static`	Initialises an area of memory for use as the C static area.
`init.heap`	Initialises an area for use as the heap area.
`terminate.static.use`	Terminates static usage.
`terminate.heap.use`	Terminates heap usage.

init.static

`PROC init.static([]INT static.area, INT required.size, gsb)`

`init.static` is used to set aside and initialise an area of memory for use as a C static area. The static area reserved is an integer array which is declared in the calling occam program.

Two integer values are returned:

`required.size` : The number of words of static space re-
quired.

`gsb` :   A pointer to the base of the array which will
act as the global static base.

Note: the number of words of static space required is equivalent to the required size of the integer array. One element of the integer array is equivalent to one word of memory.

If an error occurs on initialising the static area the value MOSTPOS INT is re-turned instead of the required size.

The procedure can be used to check the size of static area required by checking the first return integer. For example:

```
INT required.size, gsb:
[STATIC.SIZE]INT static.area:

init.static(static.area, required.size, gsb)
IF
  required.size > STATIC.SIZE
    -- not enough space reserved
  TRUE
    -- array is big enough
```

Another possible way of using `init.static` is to reserve a large amount of memory for use by the C function. To do this an initial call to `init.static` would be made with an array size of zero to obtain the required size, followed by a second call which would set up a segment of memory as the static area. The rest of the memory could be used by the occam program for its own purposes, perhaps to allocate the C heap. For example:

```
INT required.size, gsb:
[VERY.BIG.NUMBER]INT memory:

init.static([memory FROM 0 FOR 0], required.size, gsb)
```

```
static.area IS [memory FROM 0 FOR required.size]:
memory.left IS [memory FROM required.size FOR
                (VERY.BIG.NUMBER - required.size)]:
SEQ
  init.static(static.area, required.size, gsb)
  -- rest of program
```

init.heap

PROC init.heap(VAL INT gsb, []INT heap.area)

init.heap is used to set aside an area of memory for use as a C heap. The first argument is the gsb pointer returned by **init.static**, which is required because the memory allocation routines make use of static data.

Like the static area the heap area is declared as an integer array. This array must be large enough to accommodate all calls to the C memory allocation functions. The number of words of heap area required is equivalent to the size of the integer array. One element of the integer array is equivalent to one word of memory.

If the heap is used by a function before **init.heap** has been called the C memory allocation functions will fail with their normal error returns.

terminate.static.use

PROC terminate.static.use(VAL INT gsb)

terminate.static.use should be called when the static area is no longer required, usually when no further calls to C will be made. It provides a clean way of ending the use of the C static area.

Once the static terminate procedure has been called the state of the static area is undefined.

terminate.heap.use

PROC terminate.heap.use(VAL INT gsb)

terminate.heap.use should be called when the heap is no longer required. It provides a clean way of terminating the use of the heap.

Once the heap terminate procedure has been called the state of the heap is undefined.

terminate.heap.use must be called *before* terminating the static area be-

cause the heap is accessed using static variables.

9.2.5   Example of using the callc library

The following example illustrates how the callc library procedures can be used to set up and terminate the static and heap areas for a C function.

C function to be called:

```c
#include <stdlib.h>

static int *c;

int c_subfunc(int x, int check)
{
  int i;

  if (check == 0)
  {
    c = (int *)malloc(x);
    if (c == NULL)
      return 1;
    else
    {
      for (i=0; i < x / sizeof(int); i++)
        c[i] = i;
    }
  }
  else
  {
    for (i=0; i < x / sizeof(int); i++)
      if (c[i] != i)
        return 1;
  }
  return 0;
}
```

Calling occam code:

```occam
#INCLUDE "hostio.inc"
#USE "hostio.lib"
#USE "callc.lib"  -- the 'calling C' functions.

-- we cannot use the name c_subfunc because it is illegal in
-- occam therefore we translate it into a legal occam name

#PRAGMA TRANSLATE csubfunc "c_subfunc"
```

```
-- declare the C function as an occam descriptor. Note the
-- following:
-- 1. We use the translated name - csubfunc.
-- 2. We declare the gsb as the first parameter. This
--    parameter is hidden in C.
-- 3. The parameter and return types are matched to those
--    in C.
-- 4. The workspace requirement, in this case 200, is an
--    overestimation.

#PRAGMA EXTERNAL "INT FUNCTION csubfunc(VAL INT gsb, x,
                                        check) = 200"

PROC test(CHAN OF SP fs, ts, []INT freemem)
  INT length, gsb, required.size:
  -- static.area and heap.area sizes are overestimations
  VAL static.size IS 4000:
  VAL heap.size IS 4000:
  [static.size]INT static.area:
  [heap.size]INT heap.area:
  SEQ
    -- set up static.area as the static area
    init.static(static.area, required.size, gsb)
    -- now check for error
    IF
      required.size > static.size
        so.write.string.nl(fs, ts,
                "error initialising static*n")
      TRUE
        INT fail:
        SEQ
          -- set up the heap area. Note that gsb is the
          -- first parameter
          init.heap(gsb, heap.area)
          -- call the c function. Note that the gsb is
          -- passed as the first parameter. This call
          -- of csubfunc mallocs 2000 bytes and fills
          -- this area with known values.
          fail := csubfunc(gsb, 2000, 0)
          -- check for error
          IF
            fail = 0
              SEQ
                -- call csubfunc again but this time ask
                -- it to check the area set up by the
                -- previous call
                fail := csubfunc(gsb, 2000, 1)
                -- check for error
```

```
              IF
                fail = 0
                  so.write.string.nl(fs, ts,
                              "successful test*n")
                TRUE
                  so.write.string.nl(fs, ts, "FAIL*n")
            TRUE
              -- the first call to csubfunc failed
              so.write.string.nl(fs, ts,
                        "malloc FAILED*n")
  -- terminate use of heap cleanly
  terminate.heap.use(gsb)
  -- terminate use of static cleanly
  terminate.static.use(gsb)
  so.exit(fs, ts, sps.success)
:
```

9.2.6   Linking the program

The occam program must be linked with the compiled C function, the callc library, the reduced runtime C library, and with any other occam libraries it uses. In the above example the set of files to be linked would be as follows:

`callc.tco`	– compiled occam program
`csubfunc.tco`	– compiled C function
`callc.lib`	– callc library
`hostio.lib`	– occam i/o library
`libcred.lib`	– C reduced runtime library
`occama.lnk`	– linker indirect file, listing other libraries such as compiler libraries required for the linking operation. (See chapter 19 for details).

The linker allows files to either be specified on the command line or listed in an indirect file. Because there are several files required in this instance, it may be easier to supply a linker indirect file. To do this create a text file called `callc.lnk`, containing the following lines:

```
callc.tco
csubfunc.tco
callc.lib
hostio.lib
libcred.lib
#INCLUDE occama.lnk
```

The correct linker command line (using the default processor T414 in HALT mode) would be as follows:

```
ilink -f callc.lnk                          (UNIX)
ilink /f callc.lnk                     (MS-DOS/VMS)
```

The main entry point of the program is assumed to be the first entry point encountered in the input list. Details of the operation of the linker can be found in chapter 19.

9.3   Parameter passing

The following tables describe the calling conventions that must be followed when passing parameters from occam programs to imported C processes. They list the occam equivalents on 32 and 16 bit transputers for all C types. Where there is no true equivalent the action to take is given.

Formal C parameter	Actual occam parameter	
	(32 bit)	**(16 bit)**
`char` `unsigned char`	`VAL BYTE`	`VAL BYTE`
`signed char`	No direct equivalent†	No direct equivalent†
`short` `signed short`	No direct equivalent† (see Note 1)	`VAL INT` `VAL INT16`
`unsigned short`	No direct equivalent†	No direct equivalent†
`array`	(see Note 2)	(see Note 2)

`int`
`signed int`
`enum`
	`VAL INT`
	`VAL INT32`
		`VAL INT`
		`VAL INT16`

† There is no direct type equivalent in occam. Either recode the C program or pass the parameter in another form.

Note 1: A C short on a 32 bit processor is stored in 32 bits with the upper 16 bits zeroed. In occam an `INT16` on a 32 bit processor is also stored as a 32 bit value, however, in this case the upper 16 bits are ignored and not zeroed. Hence C short and occam `INT16` are not directly equivalent.

Note 2: There are two cases to be considered when passing arrays from occam to C:

(i) When all the dimensions are known, the array is passed directly e.g. occam calling code:

```
[8]INT array:
cfunc(array)
```

called C code:

```
void cfunc(int array[8]);
```

(ii) When some dimensions are unspecified occam will pass the dimensions as extra parameters following the array parameter. The C code must be written to accept these parameters e.g. occam calling code:

```
[]INT array:
cfunc (array)
```

called C code:

```
void cfunc(int array [], int arraysize);
```

Formal C parameter	Actual occam parameter	
	(32 bit)	**(16 bit)**
`unsigned int`	No direct equivalent†	No direct equivalent†
`long` `signed long`	`VAL INT` `VAL INT32`	No direct equivalent†
`unsigned long`	No direct equivalent†	No direct equivalent†
`float`	`VAL REAL32`	No direct equivalent†
`double`	No direct equivalent†	No direct equivalent†
`struct` `union`	No direct equivalent†	No direct equivalent†
`char *` `unsigned char *`	`BYTE`	`BYTE`
`signed char *`	No direct equivalent†	No direct equivalent†
`short *` `signed short *`	`INT16`	`INT16` `INT`
`unsigned short *`	No direct equivalent†	No direct equivalent†

`int *`
`signed int *`
`enum *`
	`INT`
	`INT32`
		`INT`
		`INT16`

`unsigned int *`	No direct equivalent†	No direct equivalent†
`long *` `signed long *`	`INT` `INT32`	`INT32`
`unsigned long *`	No direct equivalent†	No direct equivalent†
`float *`	`REAL32`	`REAL32`
`double *`	`REAL64`	`REAL64`
`struct *` `union *`	No direct equivalent†	No direct equivalent†
`channel *`	`CHAN`	`CHAN`

† There is no direct type equivalent in occam. Either recode the C program or pass the parameter in another form.

9.3.1 Return values

The following table outlines the conventions that must be followed when receiving occam function return values in C.

C function type	occam function type	
	(32 bit)	(16 bit)
char unsigned char	BYTE	BYTE
signed char	No direct equivalent†	No direct equivalent†
short signed short	INT16	INT INT16
unsigned short	No direct equivalent†	No direct equivalent†
int signed int enum	INT INT32	INT INT16
unsigned int	No direct equivalent†	No direct equivalent†
long signed long	INT INT32	INT32
unsigned long	No direct equivalent†	No direct equivalent†
float	REAL32	REAL32
double	REAL64	REAL64
struct union	No direct equivalent†	No direct equivalent†
Any pointer type	No direct equivalent†	No direct equivalent†

† There is no direct type equivalent in occam. Either recode the C program or return the value in another form.

9.3.2 Examples of passing parameters

The following examples shows a C function with a variety of formal parameters along with the occam code which can call it. The code for 32 bit and 16 bit transputers is given separately.

The C function to be called on a 32 bit transputer is as follows:

```
int cfunc1(int parm1);

#pragma IMS_nolink(cfunc1)      /* remove the gsb hidden
                                   parameter */

void cproc1(char c, int i,
            long l, float f,
            char *cp, short *sp,
            int *ip, long *lp,
            float *fp, double *dp,
            int array1[8],
            int array2[], int array2len);

#pragma IMS_nolink(cproc1)      /* remove the gsb hidden
                                   parameter */

int cfunc1(int parm1)
{
  return parm1 * 10;
}

void cproc1(char c, int i,
            long l, float f,
            char *cp, short *sp,
            int *ip, long *lp,
            float *fp, double *dp,
            int array1[8],
            int array2[], int array2len)
{
  int j;

  *cp = c;
  *sp = (short)c;
  *ip = i;
  *lp = l;
  *fp = f;
  *dp = (double)i;
  for (j = 0; j < 8; j++)
    array1[j] = 42;
  for (j = 0; j < array2len; j++)
    array2[j] = array2len;
}
```

The occam code to call the above C function on a 32 bit transputer is as follows:

```
#PRAGMA EXTERNAL "INT FUNCTION cfunc1(VAL INT parm1) = 100"

#PRAGMA EXTERNAL "PROC cproc1(VAL BYTE c, VAL INT i, *
                           * VAL INT32 l, VAL REAL32 f, *
                           * BYTE cp, INT16 sp, *
                           * INT ip, INT32 lp, *
                           * REAL32 fp, REAL64 dp, *
                           * [8]INT array1, []INT array2) = 100"


    BYTE c, cp:
    INT i, ip, result:
    INT16 sp:
    INT32 l, lp:
    REAL32 f, fp:
    REAL64 dp:
    [8]INT array1:
    [5]INT array2:
    SEQ
      result := cfunc1(i)
      cproc1(c, i, l, f, cp, sp, ip, lp, fp, dp, array1, array2)
```

The C function to be called on a 16 bit transputer is as follows:

```
int cfunc1(int parm1);

#pragma IMS_nolink(cfunc1)     /* remove the gsb hidden
                                  parameter */

void cproc1(char c, int i,
            short s, char *cp,
            short *sp, int *ip,
            long *lp, float *fp,
            double *dp, int array1[8],
            int array2[], int array2len);

#pragma IMS_nolink(cproc1)     /* remove the gsb hidden
                                  parameter */

int cfunc1(int parm1)
{
  return parm1 * 10;
}

void cproc1(char c, int i,
            short s, char *cp,
            short *sp, int *ip,
            long *lp, float *fp,
            double *dp, int array1[8],
            int array2[], int array2len)
{
  int j;

  *cp = c;
  *sp = s;
  *ip = i;
  *lp = (long)i;
  *fp = (float)i;
  *dp = (double)i;
  for (j = 0; j < 8; j++)
    array1[j] = 42;
  for (j = 0; j < array2len; j++)
    array2[j] = array2len;
}
```

The occam code to call the above C function on a 16 bit transputer is as follows:

```
#PRAGMA EXTERNAL "INT FUNCTION cfunc1(VAL INT parm1) = 100"

#PRAGMA EXTERNAL "PROC cproc1(VAL BYTE c, VAL INT i, *
                           * VAL INT16 s, BYTE cp, *
                           * INT16 sp, INT ip, *
                           * INT32 lp, REAL32 fp, *
                           * REAL64 dp, *
                           * [8]INT array1, []INT array2) = 100"

BYTE c, cp:
INT i, ip, result:
INT16 s, sp:
INT32 lp:
REAL32 fp:
REAL64 dp:
[8]INT array1:
[5]INT array2:
SEQ
  result := cfunc1(i)
  cproc1(c, i, s, cp, sp, ip, lp, fp, dp, array1, array2)
```

10 Low level programming

This chapter describes a number of features of the toolset occam 2 compiler which support low-level programming of transputers. These are as follows:

Allocation This allows a channel, a variable, an array or a port to be placed at an absolute location in memory.

RETYPING channels and creating channel array constructors These facilities enable channels to be manipulated.

Code insertion This allows sections of transputer machine code to be inserted into occam programs.

Dynamic code loading A set of library procedures is provided that allows an occam program to read in a section of compiled code (from a file, for example) and execute it.

Extraordinary use of links A set of library procedures is provided which allow link communications which have not completed to be handled by timeout, or be aborted by another part of the program.

Scheduling Using the predefined routine **RESCHEDULE** to reschedule processes.

Setting the error flag The transputer error flag can be explicitly set using the predefined routine **CAUSEERROR**.

10.1 Allocation

Allocation is performed using the occam **PLACE** statement, which is defined formally as follows:

allocation = **PLACE** *name* **AT** *expression* :

The **PLACE** statement in occam allows a channel, a variable, an array, or a input/output channel for a memory mapped device (**port**), to be placed at an absolute location in memory. This feature may be used for a number of purposes, for example:

- To map occam channels onto specific transputer links from within an occam program. Channels mapped onto links in this way are known as 'hard' channels.

- To map arrays onto particular hardware such as video RAM.

- To access devices (such as UARTs or latches) mapped into the trans-
  puter's address space.

10.1.1 The PLACE statement

Normally the PLACE statement should not be used to force critical arrays or
variables into on-chip RAM. The occam compiler allocates memory according
to the scheme outlined in part 2, appendix D, and cannot allow data to be placed
arbitrarily in memory. To make the best use of on-chip RAM use separate vector
space as described in section 4.7.

The address of a placed object is derived by treating the value of the expression
as a word offset into memory. In occam addresses start at zero, while physical
machine addresses start at MOSTNEG INT (#80000000 on 32-bit transputers
and #8000 on 16-bit transputers). An occam address can be considered as a
subscript to an INT vector mapped onto memory. Thus the following statement
would cause chan to be allocated address #80000004 on a 32-bit transputer:

        PLACE chan AT 1:

Addresses are calculated in this way so that the transputer links can be accessed
using code that is independent of the word length. The links are mapped to
addresses 0, 1, 2...7.

Translation from a machine address to the equivalent occam address PLACE
value can be achieved by the following declaration:

        VAL occam.addr IS
                (machine.addr><(MOSTNEG INT)) >> w.adjust:

where: w.adjust is 1 for a 16-bit transputer and 2 for a 32-bit transputer.

All placed objects must be word aligned. If it is necessary to access a BYTE
object on an arbitrary boundary, or an INT16 object on an arbitrary 16-bit bound-
ary, the object must be an element of an array which is placed on a word ad-
dress below the required address. For example, to access a BYTE port called
io.register located at physical address #40000001 on a T414 the follow-
ing must be used:

        [4]PORT OF BYTE io.regs.vec :
        PLACE io.regs.vec AT #30000000 :
        io.register IS io.regs.vec[1] :

Placement may be used on transputer boards to access board control functions

mapped into the transputer's address space. For example, on the IMS B004, the subsystem control functions (**Error**, **Reset** and **Analyse**) are mapped into the address space and can be accessed from occam as placed ports. The following code will reset the subsystem on an IMS B004:

```
PROC reset.b004.subsystem()
  VAL subsys.reset   IS #20000000:  -- address 0
  VAL subsys.error   IS #20000000:  -- address 0
  VAL subsys.analyse IS #20000001:  -- address 4
  PORT OF BYTE reset, analyse, error:
  PLACE reset   AT subsys.reset:
  PLACE analyse AT subsys.analyse:
  PLACE error   AT subsys.error:
  VAL delay IS 78:   -- 5 msec delay
  TIMER clock:
  INT time:
  SEQ
    -- set reset and analyse low
    analyse ! 0 (BYTE)
    reset ! 0 (BYTE)
    reset ! 1 (BYTE)       -- hold reset high
    clock ? time
    clock ? AFTER time PLUS delay
    reset ! 0 (BYTE)       -- reset subsystem
:
```

The error and analyse functions can be controlled from occam in a similar way.

10.1.2 Allocating specific workspace locations

A number of specialised transputer instructions require specific workspace placings. For example, the instructions **POSTNORMSN**, **OUTBYTE**, **OUTWORD** and the disabling **ALT** instructions all use workspace location 0. To accommodate this the compiler supports the following allocation:

PLACE name AT WORKSPACE n:

where: n is a constant integer. (See part 2, appendix D for syntax details).

This is used to ensure that a variable is allocated a particular position within a procedure or function's workspace. The compiler ensures that at least n words of workspace are allocated, and that no other variables are placed at that address. The compiler will warn if a variable **PLACED AT WORKSPACE n** is in scope when its own workspace allocation requires to use that workspace location, or when another is **PLACED** at the same location.

For example on a `T414`, the `POSTNORMSN` instruction can be used to pack a floating point number; it requires an exponent to be previously stored at workspace offset 0. The following code may be used:

```
REAL32 FUNCTION pack (VAL INT guard, frac, exp,
sign)
  REAL32 result :
  VALOF
    INT temp :
    PLACE temp AT WORKSPACE 0 :
    SEQ
      temp := exp
      ASM
        LDAB guard, frac
        NORM
        POSTNORMSN
        ROUNDSN
        LDL sign
        OR
        ST result
    RESULT result
  :
```

(For the background on this example, see the *Transputer instruction set – a compiler writer's guide*, section 7.11.2). Use of the `ASM` construct is described in section 10.3.1.

10.1.3 Allocating channels to links

When mapping channels to specific transputer links, the channel word is placed at the specified address for scalar channels. Arrays of channels, however, are mapped as arrays of pointers to channels :

PLACE scalar channel AT n:

places the channel word at that address.

PLACE array of channels AT n:

places the array of pointers at that address.

Note: that the current implementation of arrays of channels has changed from the IMS D705/D605/D505 releases of the toolset. In the past they were implemented as a pointer to an area of memory which held a number of contiguous channels. The data type of a channel has been changed from *'channel'* to *'pointer to channel'*. This means that code compiled by IMS D705/D605/D505 toolsets

cannot be called by code compiled with the current toolset, if channel arrays are used.

The following two code fragments illustrate the placement of channels on links.

```
CHAN OF ANY    in.link0, out.link0 :
CHAN OF ANY    in.link1, out.link1 :
CHAN OF ANY    in.link2, out.link2 :
CHAN OF ANY    in.link3, out.link3 :
CHAN OF ANY    in.event :

PLACE    out.link0 AT link0.out:
PLACE    in.link0  AT link0.in:

PLACE    out.link1 AT link1.out:
PLACE    in.link1  AT link1.in:

PLACE    out.link2 AT link2.out:
PLACE    in.link2  AT link2.in:

PLACE    out.link3 AT link3.out:
PLACE    in.link3  AT link3.in:

PLACE    in.event  AT event.in:
```

or:

```
CHAN OF ANY out.link0, out.link1, out.link2, out.link3 :
PLACE out.link0 AT link0.out :
PLACE out.link1 AT link1.out :
PLACE out.link2 AT link2.out :
PLACE out.link3 AT link3.out :
[4]CHAN OF ANY outlink IS [out.link0, out.link1,
                           out.link2, out.link3] :

CHAN OF ANY in.link0, in.link1, in.link2, in.link3 :
PLACE in.link0 AT link0.in :
PLACE in.link1 AT link1.in :
PLACE in.link2 AT link2.in :
PLACE in.link3 AT link3.in :
[4]CHAN OF ANY inlink IS [in.link0, in.link1, in.link2,
                          in.link3] :
```

Link addresses are defined in the include file `linkaddr.inc` that is supplied with the toolset.

Although shown here as `CHAN OF ANY` channels you should use specific occam channel protocols wherever possible to ensure that channels are properly checked at compile time.

10.2   RETYPING channels and creating channel array constructors

Channels may be RETYPEd. This allows the user to change the protocol on a channel in order to pass it as a parameter to another routine, for example:

```
PROTOCOL PROT32 IS INT32 :
PROC p (CHAN OF INT32 X)
  X ! 99(INT32)
:
PROC q1 (CHAN OF PROT32 y)
  SEQ
    p (y)       -- this is illegal
    CHAN OF INT32 z RETYPES y :
    p(z)        -- this is legal
:
```

The facilities for RETYPEing channels should only be used by programmers who understand the implementation of transputer channels, and the implications of attempting to circumvent occam's checking of channel usage. These facilities may be useful for those programmers who are using occam at a very low level, for example, writing loaders and other operating system type functions.

The current implementation of channels allows flexible use of channel arrays, which are implemented as an array of pointers to channel words. This means, for example, that it is possible to create an array of channels which map onto the hard links in a different order than 0 to 3, by using channel array constructors. For example:

```
CHAN OF ANY out.link0, out.link1, out.link2,
out.link3 :
PLACE out.link0 AT link0.out :
PLACE out.link1 AT link1.out :
PLACE out.link2 AT link2.out :
PLACE out.link3 AT link3.out :
[4]CHAN OF ANY outlink IS [out.link3, out.link1,
out.link2, out.link0] :
```

A particular effect of this implementation is that it may be useful to retype channels and arrays of channels into integers, in order to give the programmer access to these pointers. A programmer may set up an array of integers whose values are the addresses of channel words, and then use these as addresses of channels, like so:

```
[n] INT x:
SEQ
  ... initialise elements of array x, then:

  [n] CHAN OF protocol c RETYPES x:
  SEQ
    ... then communicate on c[i]
```

This will use the contents of x[i] as the address of the channel word. **Note:** channels set up in this way are not initialised automatically; you should initialise the contents of the channel word to MOSTNEG INT yourself, unless the channel word is mapped to a hard link.

Similarly channels may be retyped into pointers:

```
[n] CHAN OF protocol c :
SEQ
  VAL [n] INT x RETYPES c:
  SEQ i = 0 FOR n
    SEQ
      so.write.string (fs, ts, "The address of the
                               channel word of c[")
      so.write.int (fs, ts, i, 0)
      so.write.string (fs, ts, "] is : ")
      so.write.hex.int (fs, ts, x[i], 8)
      so.write.nl (fs, ts)
```

Note: retyping channels to pointers must be a **VAL RETYPE**. You may not modify the values of the pointers.

Single channels may be **RETYPED** to and from **INT**s.

Channel retyping should not be used to create arrays of existing channels. Channel array constructors may be used for this:

```
PROC fancy.mux ([2]CHAN OF INT in, CHAN OF INT
                spare, out)
  [3]CHAN OF INT c IS [in[0], in[1], spare] :
  WHILE TRUE
    ALT i = 0 FOR 3
      INT data :
      c[i] ? data
        out ! data
:
```

10.3   Code insertion

This section describes the facilities provided by the occam 2 compiler code insertion mechanism.

The code insertion mechanism enables the user to access the instruction set of the transputer directly within the framework of an occam program. Symbolic access to occam variable names is supported, as is automatic jump sizing. More details on the instruction set may be found in '*The transputer instruction set: a compiler writer's guide*'.

Code insertion may be employed to perform tasks which are not possible in occam, or for particularly time-critical sections of a program. There are two reasons, however, why code insertion should be avoided as a solution to problems which may, with some thought, be solved using occam.

The first and most important reason is that the validity of a system consisting entirely of occam can be checked by the compiler. The compiler can check usage of channels, access to variables, communication protocols and range violations, and a single code insert prevents the compiler from performing these checks adequately. A second reason is that the transputer instruction set is optimised for high level languages, particularly occam, and algorithms which are simple to code and easy to debug in occam may become difficult and obscure when coded in the transputer instruction set directly.

10.3.1   Using the code insertion mechanism

Code insertions may be introduced by either the **ASM** or **GUY** constructs. This section describes the use of the **ASM** construct. (Details of the syntax are given in part 2, appendix D).

The **GUY** construct is maintained to provide compatibility with the IMS D705, D605, D505 toolsets. Appendix B (in part 2) outlines the differences between **ASM** and **GUY** constructs. It also describes the restrictions placed on the use of the **GUY** construct by the current compiler.

The context of the **ASM** construct is determined, as with all occam constructs, by the text indentation. The transputer instructions which follow the **ASM** must be indented and there can only be one instruction per line. Lines may be terminated by a comment, which is introduced by a double dash ('--') as in occam. The transputer instructions are upper case versions of the standard mnemonics listed in '*The transputer instruction set: a compiler writer's guide*'.

Compiler options determine which instructions may be used within sections of code insertions, in the unit being compiled. The default is to disallow all code

inserts. If the '**G**' option is used, then the instructions allowed are a restricted set of instructions which are sufficient for time-critical sections of sequential code. If the '**W**' option is used, then all transputer instructions are allowed. Since the inclusion of some instructions may have an unexpected effect on the occam program (for example, instructions which move the workspace pointer), instructions outside of the restricted set must be used with great care. Transputer instructions in the restricted set are listed in part 2 appendix B.

ASM statements can contain any number of primary or secondary transputer operations, or transputer pseudo-operations or labels.

In the transputer instruction set primary operations are *direct* instructions, *prefixing* instructions, or the special indirect instruction *opr*. Primary operations are always followed by an operand which can be any constant or constant expression. If additional **pfix** or **nfix** instructions are required to encode large values the **ASM** assembler automatically generates the required bytes.

Secondary operations are any transputer *operation*, that is, any instruction selected using the *opr* instruction.

Pseudo-operations are more complex operations built up from sequences of instructions. Like macros, they expand into one or more transputer instructions, depending on their context and parameters.

For example, to perform a 1's complement addition we can write the following occam:

```
INT carry, temp:
SEQ
  carry, temp := LONGSUM (a, b, 0)
  c := carry PLUS temp
```

However, if this occurs in a time-critical section of the program we might replace it with:

```
ASM
  LDABC a, b, 0
  LSUM
  SUM
  ST c
```

which would avoid the storing and reloading of **carry** and **temp**.

Values in the range **MOSTNEG INT** to **MOSTPOS INT** may be used as operands to all of the direct functions without explicit use of prefix and negative prefix instructions. Access to non-local occam symbols is provided without explicit indirection, if you use the pseudo-instructions '**LD**', '**LDAB**' etc.

A more complex example, which sets an error if a value read from a channel is not in a particular range, takes advantage of both these facilities:

```
INT   a :
...  other code
PROC get.and.check.index (CHAN OF INT c)
  SEQ
    c ? a
    ASM
      LDAB 512, a     -- push value of free
                      -- variable onto stack
                      -- followed by 512
      CCNT1           -- if NOT (0 < a <= 512)
                      -- then set error
  :
```

If there is a requirement for the code insertion to use some work space, then the work space may be declared before the **ASM** construct, in which case, the work space locations are accessed like any other occam symbol.

```
INT   a :
SEQ
  INT   b, c :
  ASM
    LD    a       -- push value in a onto stack
    ST    b       -- pop value from stack into b
  ...  more code
```

10.3.2 Special names

The following special names are available as constants inside **ASM** expressions.

.WSSIZE   Evaluates to the size of the current procedure's workspace. This will be the workspace offset of the return address, except within a replicated **PAR**, where it will be the size of that replication's workspace requirement.

.VSPTR   Evaluates to the workspace offset of the vector space pointer. When it is used inside a replicated **PAR**, it points to the vector space pointer for that branch only. A compile time error is generated if there is no vector space pointer because no vectors have been created.

.STATIC   Evaluates to the workspace offset of the static link. When it is used inside a replicated **PAR** it points to the static link for that branch only. A compile time error is generated if there is no static link.

For example, to determine the return address of a procedure, the following could be used: `LDL .WSSIZE`.

It is not checked that these names are used sensibly, for example, `J.WSSIZE` is legal even though it has no useful effect.

10.3.3 Labels and jumps

To insert a label into the sequence of instructions, put the name of the label, preceded by a colon, on a line of its own. When the label is used in an instruction, the name is again preceded by a colon. For example:

```
ASM
    ...  some instructions
    :FRED
    ...  some more instructions
    CJ  :FRED
```

Branches may only be made to a label defined within the same procedure or function. The same label name may not be defined more than once within an occam procedure.

10.3.4 Programming notes

1 Floating-point (fp) registers cannot be loaded directly; they must be loaded or stored by first loading a pointer to the register into an integer register and then using the appropriate floating-point instruction.

2 The operands to the load pseudo-ops must be small enough to fit in a register and the operands to the store pseudo-ops must be word-sized modifiable *elements*.

10.4   Dynamic code loading

The toolset compiler permits the dynamic loading and execution of code using the procedures described in this section.

These procedures are provided automatically by the compiler and are *not* referenced by a `#USE` directive. The procedures allow you to write an occam program that reads in a compiled occam procedure, and then calls it. The called procedure may be compiled and linked separately from the calling program and read in from a file. It is possible to pass parameters to the procedure, which must have at least 3 formal parameters.

(**Note** that if you wish to dynamically load occam FUNCTIONS, it is recommended that you call the FUNCTION indirectly from an occam PROC, and use non-**VAL** parameters to return the results to the calling environment).

The procedures for setting up parameters before the call and for making the call are outlined in the table below, and described in the following sections, with examples. Further information and examples of this technique can be found in section 5.3.5 of *The Transputer Applications Notebook – Systems and Performance*.

Procedure	Parameter Specifiers

`KERNEL.RUN`	`VAL []BYTE code,`
	`VAL INT entry.offset,`
	`[]INT workspace,`
	`VAL INT`
	`no.of.parameters`

`LOAD.INPUT.CHANNEL`	`INT here,` `CHAN OF ANY in`
`LOAD.INPUT.CHANNEL.VECTOR`	`INT here,` `[]CHAN OF ANY in`
`LOAD.OUTPUT.CHANNEL`	`INT here,` `CHAN OF ANY out`
`LOAD.OUTPUT.CHANNEL.VECTOR`	`INT here,` `[]CHAN OF ANY out`
`LOAD.BYTE.VECTOR`	`INT here,` `VAL []BYTE bytes`

The bootstrap tool `icollect` described in chapter 12, can produce code in a format suitable for dynamic loading. The file format is described in chapter 12.

10.4.1 Calling code

The occam 2 compiler recognises calls of a procedure `KERNEL.RUN` with the following parameters:

```
PROC KERNEL.RUN (VAL []BYTE code,
                 VAL INT entry.offset,
                 []INT workspace,
                 VAL INT no.of.parameters)
```

The effect of this procedure is to call the procedure loaded in the `code` buffer, starting execution at the location `code[entry.offset]`.

The **code** to be called must begin at a word-aligned address. To ensure proper alignment either start the array at zero or realign the code on a word boundary before passing it into the procedure.

The **workspace** buffer is used to hold the local data of the called procedure. For details of the contents of the **workspace** buffer see figure 10.1. The required size of this buffer and the code buffer must be derived from information in the code file.

The parameters passed to the called procedure should be placed at the top of the **workspace** buffer by the calling procedure before the call of **KERNEL.RUN**. The call to **KERNEL.RUN** returns when the called procedure terminates. If the called procedure requires a separate vector space, then another buffer of the required size must be declared, and its address placed as the last parameter at the top of **workspace**. As calls of **KERNEL.RUN** are handled specially by the compiler it is necessary for **no.of.parameters** to be a constant known at compile time and to have a value ≥ 3.
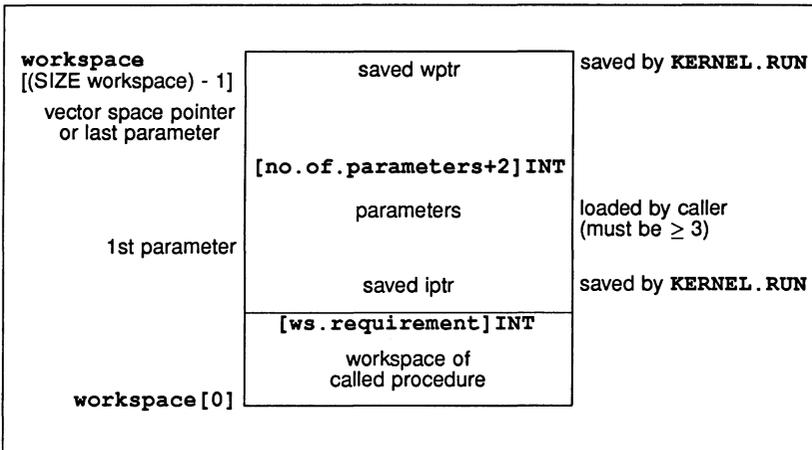
Figure 10.1 Workspace buffer

The workspace passed to **KERNEL.RUN** must be at least:

[ws.requirement + no.of.parameters + 2] INT

where **ws.requirement** is the size of workspace required, determined when the called procedure was compiled, and stored in the code file and **no.of.parameters** includes the vector space pointer if it is required.

The parameters must be loaded before the call of **KERNEL.RUN**. The parameter

corresponding to the first formal parameter of the procedure should be in the word adjacent to the saved `Iptr` word, and the vector space pointer or the last parameter should be adjacent to the top of workspace where the `Wptr` word will be saved.

Note: code developed with the current toolset will not be able to call code compiled by IMS D705/D605/D505 toolsets, if channel arrays are used. See section 10.1.3.

10.4.2   Loading parameters

There are a number of library procedures to set up parameters before the call. These are:

`LOAD.INPUT.CHANNEL (INT here, CHAN OF ANY in)`

The variable `here` is assigned the address of the input channel `in`.

`LOAD.INPUT.CHANNEL.VECTOR (INT here,`
`                           []CHAN OF ANY in)`

The variable `here` is assigned the address of the base element of the channel array `in` (i.e. the base of the array of pointers). **Note** this is a change from the previous implementation of this procedure in the IMS D705/D605/D505 toolsets which used to return the actual address of the input channel array.

`LOAD.OUTPUT.CHANNEL (INT here, CHAN OF ANY out)`

The variable `here` is assigned the address of the output channel `out`.

`LOAD.OUTPUT.CHANNEL.VECTOR (INT here,`
`                            []CHAN OF ANY out)`

The variable `here` is assigned the address of the base element of the channel array `out` (i.e. the base of the array of pointers). **Note** this is a change from the previous implementation of this procedure in the IMS D705/D605/D505 toolsets which used to return the actual address of the output channel array.

`LOAD.BYTE.VECTOR (INT here, VAL []BYTE bytes)`

The variable `here` is assigned the address of the byte array `bytes`.

Note that when passing vector parameters, if the formal parameter of the PROC called is unsized then the vector address must be followed by the number of elements in the vector, for example:

```
LOAD.BYTE.VECTOR(param[0], buffer)
param[1] := SIZE buffer
```

Thus an unsized vector parameter requires 2 parameter slots. The size must be in the units of the array (not in bytes, unless it is a byte vector, as above). For multi-dimensional arrays, one parameter is needed for each unsized dimension, in the order that the dimensions are declared.

All variables and arrays should be retyped to byte vectors before using LOAD.BYTE.VECTOR to obtain their addresses, using a retype of the form:

```
[]BYTE b.vector RETYPES variable:
```

LOAD.BYTE.VECTOR should also be used to set up the address of the separate vector space.

10.4.3 Examples

This section gives two examples of dynamic loading. The first is a simple example showing how parameterless code can be input on a channel and loaded. The second is a more complex example showing how to set up and pass parameters into a dynamically loaded program.

Example 1: load from link and run

This is a simple procedure to load a (parameterless) code packet from a link and run it. The type of the packet is given by the protocol:

```
PROTOCOL CODE.MESSAGE IS INT::[]BYTE; INT; INT
```

The code is sent first, as a counted array, followed by the entry offset and

workspace size.

```
PROC run.code (CHAN OF CODE.MESSAGE input,
                   []INT run.vector, []BYTE code.buffer)
  VAL no.parameters IS 3 :    -- smallest allowed
  INT code.length,entry.offset,work.space.size:
  INT total.work.space.size:
  SEQ
    input ? code.length::code.buffer;
              entry.offset; work.space.size
    total.work.space.size :=
       (work.space.size + no.parameters) + 2
    []INT   work.space IS [run.vector FROM 0 FOR
                         total.work.space.size] :
    KERNEL.RUN (code.buffer, entry.offset,
                work.space, no.parameters)
  :
```

Example 2: system loader

This example shows how to set up parameters prior to running code loaded from a file. It is assumed that the code requires use of a separate vector space.

Consider a process with an entry of the form:

```
PROC process (CHAN OF ANY fs, ts, []INT buffer,
              VAL BOOL debugging, INT result)
```

The two channel parameters `fs` and `ts` handle output from and input to the file server; the `INT` vector acts as a buffer. The two channels and the buffer are the same parameters as are provided by the bootstrap code added by the `collector` tool (chapter 12), and the example takes advantage of this. The fourth parameter is a value parameter that will not be changed by the process, so only the value needs to be passed. The final parameter is an `INT` that will be changed by the process, and its address must be passed into the procedure.

The calling program is shown below. The program reserves 256 bytes for the code that is to be read in; if you use this program make sure you modify this value to suit the size of your own code.

```
PROC call.program (CHAN OF ANY fs, ts, []INT free.memory)

  -- Variables for holding code and entry and workspace
  -- data read from file
  [256]BYTE code:
  INT code.length, entry.offset, work.space.size:
  INT vector.space.size:
  INT result:   -- Variable used by process
  VAL debugging IS TRUE:   -- Value param for process
```

```
VAL no.params IS 7:  -- No. of parameter slots
-- Need 1 slot per parameter + 1 for the size of the
-- array parameter + 1 for the vector space pointer

SEQ
  ...   Read in code and data about code

  -- Slice up memory vector for use by process
  []INT ws IS [free.memory FROM 0 FOR
          (work.space.size PLUS 3) PLUS no.params]:
  -- Reserve work space requirement for process
  []INT parameter IS [ws FROM work.space.size PLUS
                      1 FOR no.params]:
  -- Reserve slot in ws for parameters
  []INT vs IS [free.memory FROM SIZE ws FOR
                vector.space.size]:
  -- Reserve vector space requirement for process
  []BYTE b.vs RETYPES vs:
  -- Retype as a byte vector
  -- All vectors must be loaded as byte vectors.
  []INT buffer IS [free.memory FROM (SIZE ws) PLUS
                    (SIZE vs) FOR
                    (SIZE memory) MINUS ((SIZE ws)
                    PLUS (SIZE vs))]:
  -- Reserve remainder of memory for use
  -- as process parameter buffer
  []BYTE b.buffer RETYPES buffer:
  -- Retype as a byte vector
  []BYTE b.result RETYPES result:
  -- All variables must be retyped as a byte vector
  SEQ
    LOAD.INPUT.CHANNEL(parameter[0], fs)
    LOAD.OUTPUT.CHANNEL(parameter[1], ts)
    LOAD.BYTE.VECTOR(parameter[2], b.buffer)
    parameter[3] := SIZE buffer
    parameter[4] := INT debugging
    -- Store value parameter
    LOAD.BYTE.VECTOR(parameter[5], b.result)
    -- Load address of INT parameter
    LOAD.BYTE.VECTOR(parameter[6], b.vs)
    -- set pointer to vector space
    KERNEL.RUN([code FROM 0 FOR code.length],
              entry.offset, ws, no.params)
    -- Run the process
:
```

This example first declares the variables and constants required for the pro-
cess. The vector `code` should be of a size large enough to hold the code for
the process. The values of the variables `code.length`, `entry.offset`,

work.space.size and **vector.space.size** are determined from the data in the code file.

Next the vector **free.memory** is partitioned for use as the process's work space, vector space and as the variable vector used by the process. All vectors and variables used by the process must be retyped as byte vectors so that their address can be determined by the predefined routine **LOAD.BYTE.VECTOR**.

The parameters for the process are then set up. The unsized vector **buffer** is passed as an address followed the size of the vector, in integers. Note that the size of **buffer**, not **b.buffer**, is used.

The partitioning of the free memory buffer is illustrated in figure 10.2.
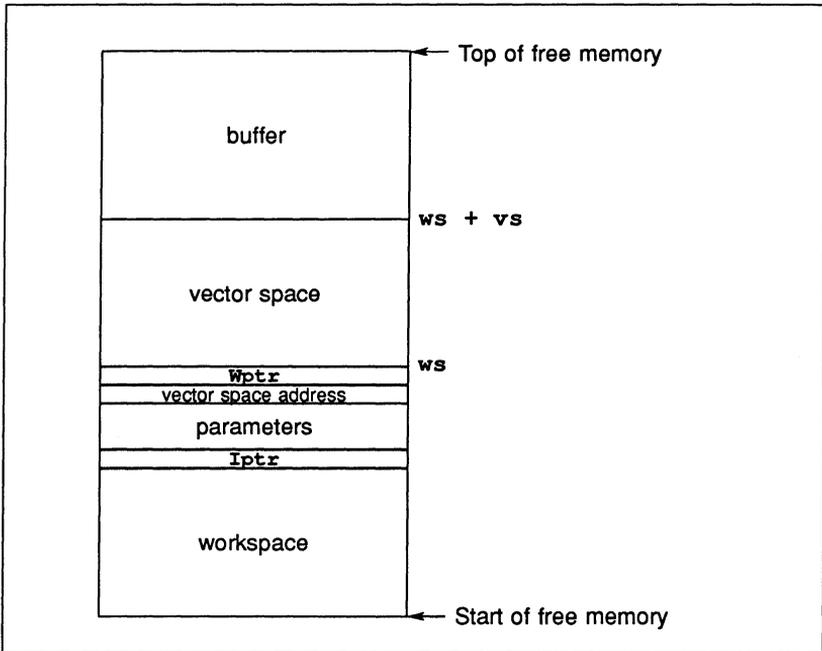
Figure 10.2 Partitioning of free memory

10.5   Extraordinary use of links

Introduction

The transputer link architecture provides ease of use and compatibility across the range of transputer products. It provides synchronised communication at the message level which matches the occam model of communication.

In certain circumstances, such as communication between a development system and a target system, it is desirable to use a transputer link even though the synchronised message passing of occam is not exactly what is required. Such extraordinary use of transputer links is possible but requires careful programming and the use of some special occam procedures.

The use of these procedures is described in this chapter. To use them in a compilation unit, the directive #USE "xlink.lib" should be inserted at the top of the source for that unit. For details of the procedures see part 2, section 1.9.

10.5.1   Clarification of requirements

As an example, consider a development system connected via a link to a target system. The development system compiles and loads programs onto the target and also provides the program executing in the target with access to facilities such as a file store. Suppose the target halts (because of a bug) whilst it is engaged in communication with the development system. The development system then has to analyse the target system.

A problem will arise if the development system is written in 'pure' occam. It is possible that when the target system halts, the development system is in the middle of communicating on a link. As a result, the input or output process will not terminate and the development system will be unable to continue. This problem can occur even where an input occurs in an alternative construct together with a timeout (as illustrated below). When the first byte of a message is received the process performing the alternative is committed to input; the timer guard cannot subsequently be selected. Hence, if insufficient data is transmitted the input will not terminate.

```
ALT
  TIME ? AFTER timeout
    . . .
  from.other.system ? message
    . . .
```

It is important to note that the problem arises from the need to *recover* from the communication failure. It is perfectly straightforward to *detect* the failure within

'pure' occam and this is quite sufficient for implementing resilient systems with multiple redundancy.

10.5.2   Programming concerns

The first concern of a designer is to understand how to recognise the occurrence of a failure. This will depend on the system; for example, in some cases a timeout may be appropriate, in others the failure may need to be signalled to another process on a channel.

The second concern is to ensure that even if a communication fails, all input processes and output processes will terminate. As this cannot be achieved directly in occam, there are a number of library procedures which perform the required function. These are described below.

The final concern is to be able to recover from the failure and to re-establish communication on the link. This involves reinitialising the link hardware; again there is a suitable library procedure to allow this to be performed.

10.5.3   Input and output procedures

There are four library procedures which implement input and output processes which can be made to terminate even when there is a communication failure. They will terminate either as the result of the communication completing, or as the result of the failure of the communication being recognised. Two procedures provide input and output where communication failure can be detected by a simple timeout, the other two procedures provide input and output where the failure of the communication is signalled to the procedure via a channel. The procedures have a boolean variable as a parameter which is set **TRUE** if the procedure terminated as a result of communication failure being detected, and is set **FALSE** otherwise. If the procedure does terminate as a result of communication failure then the link channel can be reset.

All four library procedures take as parameters a link channel **c** (on which the communication is to take place), a byte vector **mess** (which is the object of the communication) and the boolean variable **aborted**. The choice of a byte vector as the parameter to these procedures allows an object of any type to be passed along the channel provided it is retyped first. Channel retyping (see section 10.2) may be used to pass channels of any protocol to these procedures.

The two procedures for communication where failure is detected by a timeout take a timer parameter **TIME**, and an absolute time **t**. The procedures treat the communication as having failed when the time as measured by the timer **TIME** is **AFTER** the specified time **t**. The names and the parameters of the procedures

are as follows:

```
InputOrFail.t(CHAN OF ANY c, []BYTE mess,
              TIMER TIME,
              VAL INT t, BOOL aborted)

OutputOrFail.t(CHAN OF ANY c, VAL []BYTE mess,
               TIMER TIME,
               VAL INT t, BOOL aborted)
```

The other two procedures provide communication where failure cannot be detected by a simple timeout. In this case failure must be signalled to the inputting or outputting procedure via a message on the channel `kill`. The message is of type `INT`. The names and parameters to the procedures are as follows:

```
InputOrFail.c(CHAN OF ANY c, []BYTE mess,
              CHAN OF INT kill, BOOL aborted)

OutputOrFail.c(CHAN OF ANY c, VAL []BYTE mess,
               CHAN OF INT kill, BOOL aborted)
```

10.5.4   Recovery from failure

To reuse a link after a communication failure has occurred it is necessary to reinitialise the link hardware. This involves reinitialising both ends of both channels implemented by the link. Furthermore, the reinitialisation must be done after all processes have stopped trying to communicate on the link. So, although the `InputOrFail` and `OutputOrFail` procedures reset the link automatically when they abort a transfer, it is necessary to use the fifth library procedure `Reinitialise(CHAN OF ANY c)` after it is known that all activity on the link has ceased.

The `Reinitialise` procedure must only be used to reinitialise a link channel after communication has finished. If the procedure is applied to a link channel which is being used for communication the transputer's error flag will be set and subsequent behaviour is undefined.

10.5.5   Example: a development system

For our example consider the development system described in section 10.5.1, illustrated in figure 10.3.

The first step in the solution is to recognise that the development system knows when a failure might occur, and hence knows when it might be necessary to abort a communication.
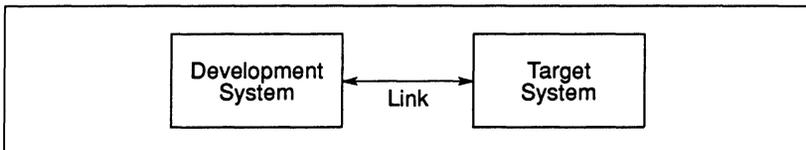
Figure 10.3 Development system

When the development system decides to reset the target it can send a message to the interface process directing it to abort any transfers in progress. It can then reset the target system (which resets the target end of the link) and reinitialise the link.

The example program below could be that part of the development system which runs when the target system starts executing and continues until the target is reset and the link is reinitialised.

```
SEQ
  CHAN OF ANY terminate.input, terminate.output :
  PAR
    ...  interface process
    ...  monitor process
    ...  reset target system
  Reinitialise(link.in)
  Reinitialise(link.out)
```

The monitor process will output on both **terminate.input** and **terminate.output** when it detects an error in the target system.

The interface process consists of two processes running in parallel; one process outputs to the link, and the other inputs from the link. As the structures of the two processes are similar only the output process is illustrated here.

If there were no need to consider the possibility of communication failure the process might be:

```
WHILE active
  SEQ
    ...
    ALT
      terminate.output ? any
        active := FALSE
      from.dev.system ? message
        link.out ! message
    ...
```

This process will loop, forwarding input from **from.dev.system** to **link.out**, until it receives a message on **terminate.output**. However,

if the target system halts without inputting after this process has attempted to forward a message, the interface process will fail to terminate.

The following program overcomes this problem:

```
WHILE active
  BOOL aborted :
  SEQ
    ...
    ALT
      terminate.output ? any
        active := FALSE
      from.dev.system ? word
        SEQ
          OutputOrFail.c (link.out, message,
                          terminate.output, aborted)
          active := NOT aborted
```

This program is always prepared to input from `terminate.output`, and is always terminated by an input from `terminate.output`. There are two possible cases. The first is where a message is received by the input which then sets `active` to `FALSE`. The second is where the output is aborted. In this case the whole process is terminated because the variable `aborted` would then be true.

10.6   Scheduling

Processes in occam may have one of two priorities, high or low. A high priority process will be executed in preference to a low priority process if both are active, so that a low priority process will be interrupted. The `PRI  PAR` construct is used to assign priority to processes.

Scheduling in occam is achieved using the transputer's scheduler which maintains a list of processes. The following predefined procedure may be used to affect scheduling:

- `RESCHEDULE ()` – inserts enough instructions into the program to cause the current process to be moved to the end of the current priority scheduling queue, even if the current process is a 'high priority' process.

10.7   Setting the error flag

The transputer error flag can be explicitly set from software using the following predefined procedure:

- **CAUSEERROR()** – inserts a `seterr` instruction into the program. If the program is in STOP or UNIVERSAL mode it inserts a `stopp` instruction as well.

CAUSEERROR(). This procedure is recognised automatically by the compiler and does not need to be referenced by the **#USE** directive.

CAUSEERROR sets the transputer error flag no matter what the error mode of the compilation. This is distinct from the occam primitive process **STOP**, which only sets the flag if the compilation is in HALT mode.

If the program was loaded using the `iserver` 'SE' option, the server terminates when the error flag becomes set.

11 EPROM programming

11.1 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 11.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.
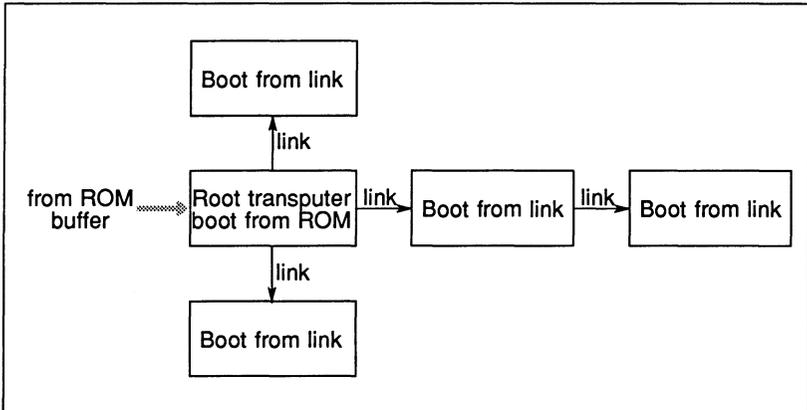
Figure 11.1 Loading a network from ROM

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.

Figures 11.2 and 11.3 illustrate the stages of preparing ROM-bootable software. Figure 11.2 shows a single occam program, compiled and linked for a single processor. Figure 11.3 shows a configured occam program, consisting of multiple linked units, connected together and allocated to processors as described

in an occam configuration file.
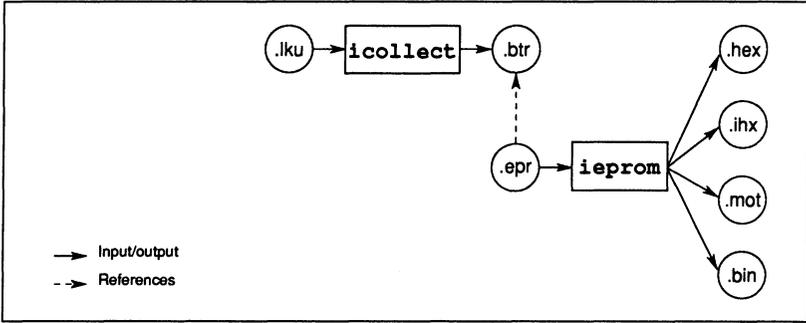
Figure 11.2 Preparation of ROM-bootable software (single processor program)
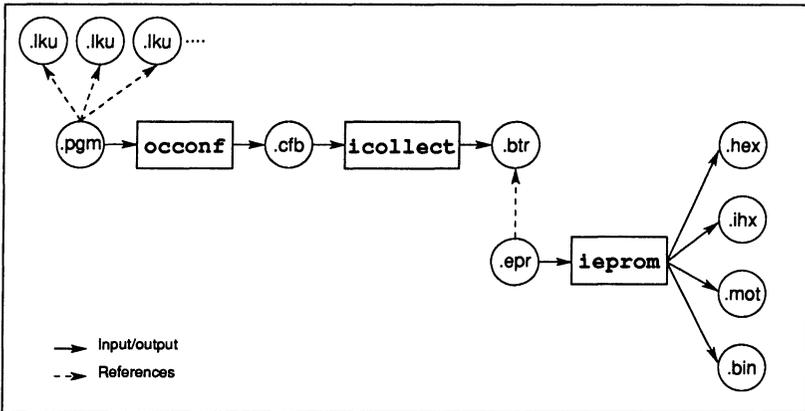
Figure 11.3 Preparation of ROM-bootable software (configured program)

11.2   Processing configurations

The processing configuration used will depend on the type of program, 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.

11.2.1   Single program, single processor, run from ROM

The application process is prepared as a single linked program. The application program is then run in the processor, directly from ROM, using the RAM as the data area for workspace and vector space.

11.2.2   Configured program, single processor, run from ROM

The application is 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.

11.2.3   Single program, single processor, run from RAM

The application is prepared as a single linked program. When booted from ROM, the processor loads the code into RAM, and executes it there; the data area is also in RAM.

11.2.4   Configured program, single processor, run from RAM

The application is described in a configuration file. When booted from ROM, the processor loads the code for the program into RAM, and sets it running, with the data area also in RAM.

11.2.5   Configured program, multiple processor, run from RAM

The application is 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 is the configuration shown in figure 11.1).

11.2.6   Configured program, multiple processor, root run from ROM, rest of network run from RAM

The application is 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.

11.3   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 config-
uration*. 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 16. The chapter also
describes `icvemit`, the tool which converts memory configuration files pro-
duced by previous toolsets i.e. the IMS D705/D605/D505 toolsets, to the format
supported by the current toolset.

`ieprom` 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 17.

11.4   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.

In addition the **NETWORK** description in the configuration file should indicate:

- which processor is the root processor, by setting its **root** attribute to **TRUE**

- the size of the ROM on that processor, by setting its **romsize** attribute to the appropriate value, in bytes.

The collector will add the appropriate ROM bootstrap to the application code and the output file will be given the extension **.btr**.

11.5   Summary of EPROM tool steps for different processing configurations

	Compile and link	Configure	Collect	EPROM
Single program, single processor, run from ROM.	Compile and link program as a single unit.	Not needed.	Collect with the **ro** and **t** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Configured program, single processor, run from ROM.	Compile and link a set of units.	Configure with the **ro** option.	Collect with the **ro** option.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single program, single processor, run from RAM.	Compile and link program as a single unit.	Not needed	Collect with the **ra** and **t** options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Configured program, single processor, run from RAM.	Compile and link a set of units.	Configure with the **ra** option.	Collect with the **ra** option.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Configured program, multiple processor, run from RAM.	Compile and link a set of units.	Configure with the **ra** option.	Collect with the **ra** option.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Configured program, multiple processor root runs from ROM, rest of network runs from RAM.	Compile and link a set of units.	Configure with the **ro** option.	Collect with the **ro** option.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

Tools

12 `icollect` — code collector

This chapter describes the code collector tool `icollect` which generates bootable or executable files for single and multitransputer programs, from linked units and configuration data files 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.

12.1 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 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 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 supervisory program.

The main inputs and outputs of the collector tool for bootable programs are shown below.

Single processor program:
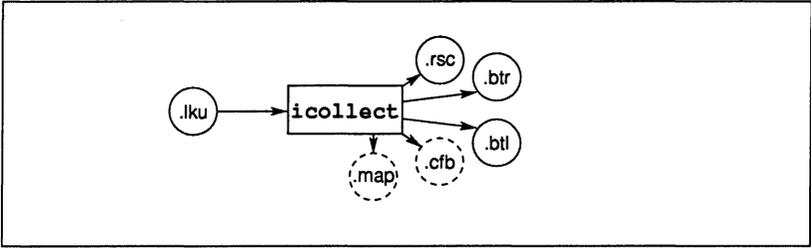
Multiple processor programs:
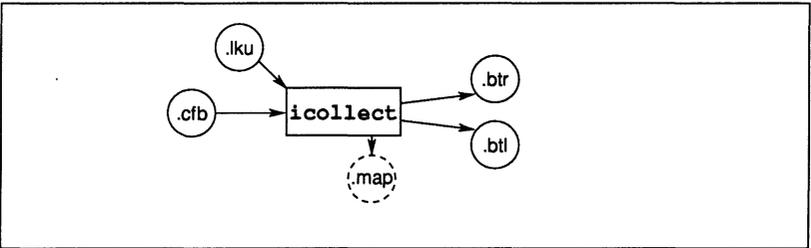
12.2   Running the code collector

The code collector is invoked using the following command line:

▶   **icollect**   *filename*   {*options*}

where: *filename* is a configuration data file created by **occonf** 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 F. 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. (See section 12.2.3).
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. This option results in a smaller amount of code being produced. (See section 12.3).
O *filename*	Specifies the output file. A filename must be supplied and is used as given. (See section 12.2.3).
P *filename*	Specifies a name for a memory map 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 `occonf` '**RA**' option.

Option	Description
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 `occonf` 'RO' option.
RS *romsize*	Specifies the size of ROM on the root processor. Only valid when used with the 'RA' or 'RO' options. *romsize* can be specified in Kilobytes and Megabytes using the 'K' or 'M' suffixes or it may be specified in hexadecimal using the '#' or '$' prefixes. *romsize* must match the romsize specified in the configuration description, if used.
S *stacksize*	Specifies the extra runtime stack size in words for single transputer programs, written in languages such as C. Can only be used with the 'T' option. *stacksize* can be specified in Kilobytes and Megabytes using the 'K' or 'M' suffixes. *stacksize* may also be specified in hexadecimal using the '#' or '$' prefixes.
T	Creates a bootable file for a single transputer. The input file specified on the command line must be a linked unit.
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.
Y	Disables interactive debugging with `idebug` and reduces the amount of memory used. (See section 12.9). Can only be used with the 'T' option.

12.2.1 Examples of use

Example A (single processor mode):

UNIX based toolsets:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
`icollect simple.lku -t`
iserver -se -sb simple.btl

MS-DOS and VMS based toolsets:

oc simple
ilink simple.tco hostio.lib /f occama.lnk
icollect simple.lku /t
iserver /se /sb simple.btl

Example B (configured program mode):

UNIX based toolsets:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
occonf simple.pgm
icollect simple.cfb
iserver -se -sb simple.btl

MS-DOS and VMS based toolsets:

oc simple
ilink simple.tco hostio.lib /f occama.lnk
occonf simple.pgm
icollect simple.cfb
iserver /se /sb simple.btl

12.2.2 Input files

The input file is either a configuration data file generated by `occonf`, 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.

12.2.3 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

A memory map file may also be generated by specifying the 'P' command line option. The format of these files is described in section 12.5.

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.

12.2.4 Small values of IBOARDSIZE

When the 'T' is used, very small values of IBOARDSIZE (including zero) are detected at runtime and prevent the program from being run. IBOARDSIZE must be ≥ to the total memory requirements of the user program being executed. See section 12.3.

12.3   Program interface

For programs which are to be loaded onto a single processor, the program interface must conform to the appropriate format, according to whether both the 'T' and 'M' options or just the 'T' is specified.

12.3.1 Interface used for 'T' option

In the case where the 'T' option is used, without specifying *memorysize*, the program must conform to one of the following procedure declarations:

Note: that these procedure declarations are compatible with those required by the IMS D705/D605/D505 releases of the toolset.

```
PROC program (CHAN OF SP from.link, to.link,
              []INT user.buffer)

PROC program (CHAN OF SP from.link, to.link,
              []INT user.buffer, stack.buffer)
```

where: `from.link` and `to.link` are the input and output channels respectively of the transputer link down which the transputer was booted.

`user.buffer` is the free memory buffer.

`stack.buffer` is a buffer allocated at the base of memory by the collector, whose size is determined by the 'S' option. If the 'S' option is not specified when `icollect` is invoked this buffer will be of size zero.

The parameter `user.buffer` which is passed to the program, is a vector that represents the amount of *free* memory that is still available on the board for use by the program. That is, memory not already used by the program for its code and workspace.

To calculate the actual memory available, the loader first reads the total memory size from the host environment variable `IBOARDSIZE`. This communication with the host is performed after the program has been loaded onto the transputer board and before the program is started. The size of the free memory vector passed to the program is given by `IBOARDSIZE` minus the combined program code and workspace allocation.

The process which reads `IBOARDSIZE` requires, at present, approximately 3.5K of memory. This process is executed and terminated before the user program runs. The segment of memory used by the process is returned to the user program as free memory. Therefore when the user program executes it will not know whether the process was present or not.

When the 'M' option is used to specify the memory size, `IBOARDSIZE` is not read and therefore the amount of memory required will be approximately 3.5K less than that required in the above case.

Warning messages

While the loader is executing the initialisation process, described above, warning messages may be obtained which have the following format:

Warning -System initialisation - *message*

where: *message* can be one of the following:

Unable to read IBOARDSIZE

> **IBOARDSIZE** environment variable is not defined correctly.

Illegal format number *number*

> The value specified for **IBOARDSIZE** is in the wrong format.

Illegal 16 bit memory size. Set to zero.

> The value of **IBOARDSIZE** is greater than 64K but a 16 bit processor is being used. The memory size has therefore been set to zero.

Negative memory size, set to zero

> A negative value was specified for **IBOARDSIZE**, which has been set to zero.

Unable to reset free memory

> The loader cannot return the memory it has used, to the user.

12.3.2 Interface used for 'T' and 'M' options

In the case where both the 'T' and the 'M' options are used, the program must conform to one of the following two procedure declarations:

```
PROC program (CHAN OF any protocol from.link,
                    to.link, []INT user.buffer)
```

```
PROC program (CHAN OF any protocol from.link,
                    to.link, []INT user.buffer,
                    stack.buffer)
```

where: The channel protocol can take any valid type.

The other variables are as defined above.

12.4 Memory allocation for single processor

The memory allocation outlined in this section applies only to single processor programs collected with the 'T' option and without the 'K' option.

The default bootstrap loader attempts to optimise placement of the program's, and its own, code and workspace. The rules it uses are as follows:

1 If present **stack.buffer** is placed at the bottom of the memory. This is followed in order by the workspace, code, vector space and free memory.

2 If the program uses a separate vector space the loader reserves a portion of the program's memory as vector space. From the size of this vector space, the size of the program code, and the size of its workspace, the loader determines the offset, from the start of memory of *free* (unused) memory. This offset is used in conjunction with the environment variable **IBOARDSIZE** to determine the amount of memory available to the program, which is then passed as a vector parameter **user.buffer** for the program to use.

Figure 12.1 shows the memory map of the loaded occam code as created by the default bootstrap loader.
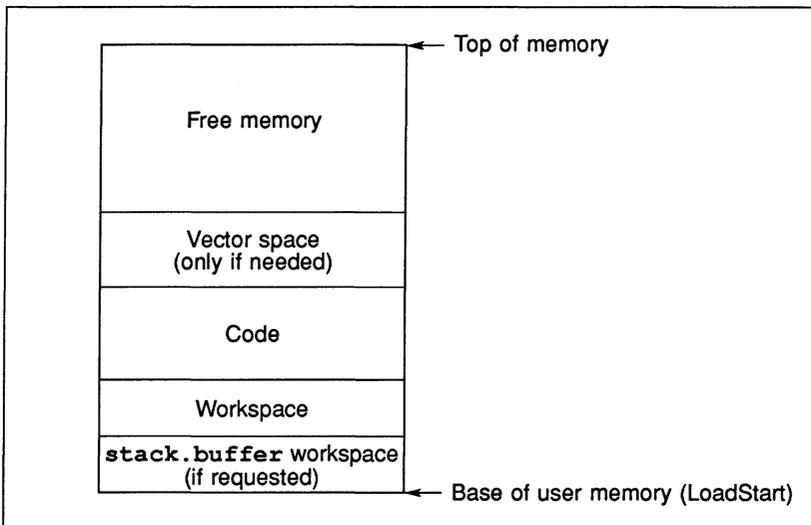
Figure 12.1 Memory map for occam programs

A memory map file may be obtained by specifying the 'P' command line option. The content of memory map files is described in section 12.5.

12.4.1 Memory allocation for mixed language programs

For mixed language programs which include modules written, for example in C, the bootstrap loader must also allocate memory for static data as well as stack and heap areas.

When the collector 'S' option is specified the **stack.buffer** placed at the bottom of memory, is used for stack by the non-occam language modules. When the 'S' option is not specified, a stack area is allocated at execution time, at the top of free memory.

Areas for static data and heap are always allocated at execution time by the non-occam language's runtime system. These areas are placed at the bottom of free memory. The heap area grows upwards, towards the top of memory and the stack grows downwards.

Figures 12.2 and 12.3 show the memory map layouts for mixed language code for programs with and without the stack requirement specified by the user.

LoadStart is described in section 12.5.

```
                                  ◄─┬ Top of memory
                                    │
                  ↑                    Free memory
               Heap
              Static                 ↓
            Vector space
           (only if needed)

               Code

          occam workspace

          stack.buffer
            workspace             ◄── Base of user memory (LoadStart)
```
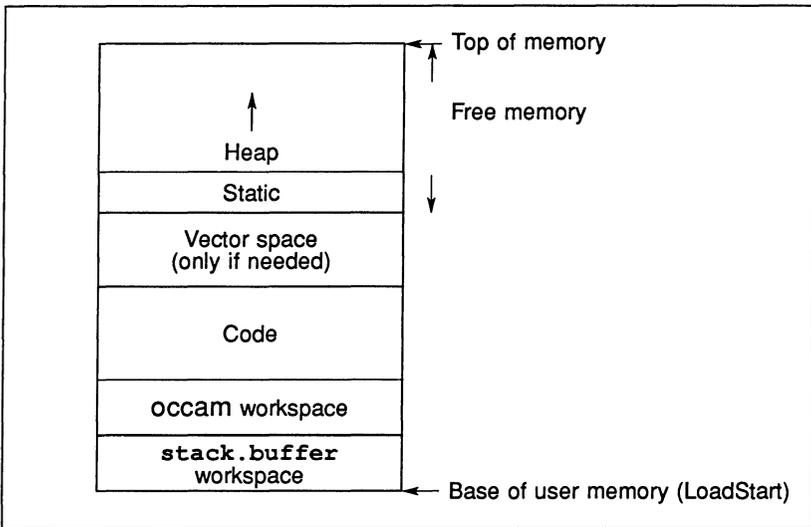
Figure 12.2 Memory map (mixed language) with stack specified
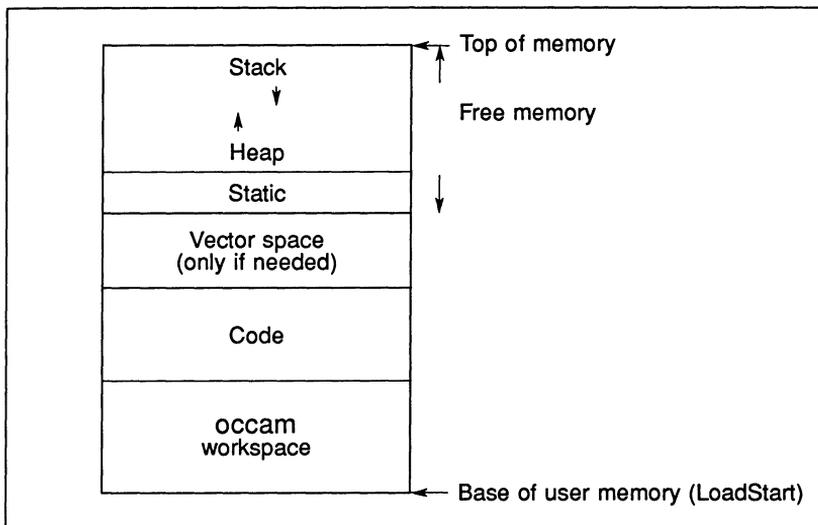
Figure 12.3 Memory map (mixed language) without stack specified

12.5   The memory map file

A memory map file may be obtained by specifying the 'P' command line option, followed by a filename. Such files contain the memory layout for each processor in the network.

The file layout takes the form of a list of code and data to be placed on respective processors. The right hand side of the file gives the start and end address followed by the size of each block.

The file contains the following information:

- **icollect** version data

- For each processor the following details are given:

    – Processor type

    – Error mode (HALT or STOP)

    – **LoadStart** (lowest user memory)

    – For each process on this processor the following is listed:

        * Code, name of file, offset from start (decimal), start ad-

dress and end address (hex), size (decimal), entry address (if any, in Hex)

* Workspace, start and end address (hex), size (decimal)

* Any other data requirements

- Boot path for the network - only present if program is configured

- Connectivity of the network - only present if program is configured

The absolute addresses are calculated using **LoadStart**, which is the base of user memory. This varies for different processor types i.e the value of **Load Start** for a T4 processor is different to that for a T8.

The memory from **MemStart** to **LoadStart** is used by the low level bootstraps and their workspace.

The addresses allocated to various data items reflect the command line options specified to the collector. Details of the memory map files for the following types of files are given below:

- Single processor, boot from link programs targetted at a specific processor type.

- Single processor, boot from link programs targetted at processor class TA or TB.

- Configured, boot from link programs.

- Boot from ROM (single and configured)

12.5.1   Single processor, boot from link

The first memory map described in this section is for a program which is to be booted for a specific processor type.

The example shown in figure 12.4 was produced by the following command line:

```
icollect -t simple.lku -p simple.map -s 400        (UNIX)
icollect /t simple.lku /p simple.map /s 400 (MS-DOS/VMS)
```

where: **simple.lku** was produced by compiling and linking the example program **simple.occ** for a T414 in the default halt on error mode.

```
icollect : INMOS toolset collector
Sun Version 2.0.25

Memory map for processor 0 T414
Load Start is 8000012C,  HALT ON ERROR
  LOW priority process 'Init.system'
    Code from 'sysproc.lib', file offset 6901
                                     #800001BC  #800003DC     544
      Entry address                 #800001BE
    Workspace                       #8000012C  #8000019C     112

  LOW priority process 'System.process.a'
    Code from 'sysproc.lib', file offset 19881
                                     #80001498  #80001DF0    2392
      Entry address                 #8000149A
    Workspace                       #80001264  #8000147C     536
    Vectorspace                     #80001DF0  #80001E70     512

  HIGH priority process 'System.process.b'
    Code from 'sysproc.lib', file offset 29562
                                     #80000410  #8000046C      92
      Entry address                 #80000411
    Workspace                       #800003DC  #800003F4      24

  LOW priority USER process
    Code from 'simple.lku', file offset 2
                                     #800008B8  #80000E8C    1492
      Entry address                 #800008B8
    Workspace                       #8000076C  #80000894     296
    Extra stack                     #8000012C  #8000076C    1600
    Vectorspace                     #80000E8C  #80000F0C     512

Parameter data                      #8000108C  #80001264     472
```

Figure 12.4 Memory map file for a single T414 processor program

LoadStart is the lowest memory location of user memory. All other addresses are calculated from **LoadStart**.

The file lists code and data segments to be placed on each processor. For each process the workspace and vector space requirements are given together with the entry point of the process. Notice that the first three processes listed are non-user processes; this will always be the case for this type of program.

Because the program was compiled with vector space enabled and the collector 'S' option was used, the user process requires the following areas of memory to be allocated:

memory area	Start address	End address
Workspace	#8000076C	#80000894
Extra stack	#8000012C	#8000076C
Vector space	#80000E8C	#80000F0C

(Normally you would only use the 'S' option for mixed language programs).

The second memory map described in this section is for a program which is to be booted for processor classed TA or TB.

The example shown in figure 12.5 was produced by the following command line:

```
icollect -t simple.lku -p simple.map          (UNIX)
icollect /t simple.lku /p simple.map     (MS-DOS/VMS)
```

where: `simple.lku` was produced by compiling and linking the example program `simple.occ` for class TA in the default halt on error mode.

```
icollect : INMOS toolset collector
Sun Version 2.0.25

Memory map for processor 0 TA
Load Start is UNKNOWN,  HALT ON ERROR
  LOW priority process 'Init.system'
    Code from 'sysproc.lib', file offset 6901
                                             #AF8      #D18      544
        Entry address                        #AFA
    Workspace                                #90       #100      112

  LOW priority process 'System.process.a'
    Code from 'sysproc.lib', file offset 19881
                                             #F4C      #18A4     2392
        Entry address                        #F4E
    Workspace                                #D18      #F30      536
    Vectorspace                              #18A4     #1924     512

  HIGH priority process 'System.process.b'
    Code from 'sysproc.lib', file offset 29562
                                             #34       #90       92
        Entry address                        #35
    Workspace                                #0        #18       24

  LOW priority USER process
    Code from 'simple.lku', file offset 2
                                             #14C      #720      1492
        Entry address                        #14C
    Workspace                                #0        #128      296
    Vectorspace                              #720      #7A0      512

Parameter data                               #920      #AF8      472
```

Figure 12.5 Memory map file for a single TA processor program

The memory layout is exactly the same as for the single processor case. However, `LoadStart`, from which the start and end addresses are calculated, can only be calculated at runtime. This is because the value of `MemStart` cannot be determined at collect time. The numbers given, in place of absolute addresses are offsets from `LoadStart`.

12.5.2   Configured program boot from link

The example shown in figure 12.6 was produced by the following command line:

```
icollect sortb3.pgm -p sorter.map            (UNIX)
icollect sortb3.pgm /p sorter.map         (MS-DOS/VMS)
```

where: `sortb3.pgm` is the configuration data file introduced in chapter 5 for the example pipeline sorter program. The other components of the program `element.occ` and `inout.occ` where compiled and linked for T414 processors in halt on error mode.

Note: only part of the file is shown in figure 12.6.

The Memory map for the configured program is similar to that produced for single transputer programs except that it has two additional sections at the end of the file. The boot path for the network lists processors in the order in which they are to be booted. The connectivity for the network lists the link connections between the processors.

```
icollect : INMOS toolset collector
Sun Version 2.0.25

Memory map for 'B003.t[0]' processor 0 T414
Load Start is 8000012C,  HALT ON ERROR

  HIGH priority process 'Init.system'
    Code from 'sysproc.lib', file offset 4969
                                       #800001C0  #800003E0      544
      Entry address                    #800001C2
    Workspace                          #8000012C  #8000019C      112

  HIGH priority process 'System.process.b'
    Code from 'sysproc.lib', file offset 28822
                                       #8000040C  #80000468       92
      Entry address                    #8000040D
    Workspace                          #800003E0  #800003F8       24

  LOW priority process 'inout.p'
    Code from 'element.lku', file offset 2
                                       #800011E4  #800013B0      460
    Code from 'inout.lku', file offset 2
                                       #800013B0  #80001A2C     1660
    Code from 'sortb3.clu', file offset 2
                                       #80001A2C  #80001C00      468
      Entry address                    #80001A2E
    Workspace                          #8000012C  #800011C0     4244
    Vectorspace                        #80001C00  #80001EC8      712

  Parameter data                       #80001EC8  #800021C4      764

Memory map for 'B003.t[1]' processor 1 T414
Load Start is 8000012C,  HALT ON ERROR

  HIGH priority process 'Init.system'
    Code from 'sysproc.lib', file offset 4969
                                       #800001C0  #800003E0      544
      Entry address                    #800001C2
    Workspace                          #8000012C  #8000019C      112

  HIGH priority process 'System.process.b'

      ....            ....                 ....         ....
      ....            ....                 ....         ....

  Parameter data                       #80001384  #80001650      716

Boot path for network

  Boot processor 0 down link 0 from HOST
  Boot processor 1 down link 3 from processor 0 link 2
  Boot processor 2 down link 3 from processor 1 link 2
  Boot processor 3 down link 2 from processor 0 link 3

Connectivity for network

  Connect HOST to processor 0 link 0
  Connect processor 1 link 3 to processor 0 link 2
  Connect processor 2 link 3 to processor 1 link 2
  Connect processor 3 link 3 to processor 2 link 2
  Connect processor 3 link 2 to processor 0 link 3
```

Figure 12.6 Memory map file for a single TA processor program

12.5.3   Boot from ROM programs

There are four cases for this type of program:

- Single processor, boot from ROM, run in RAM

- Single processor, boot from ROM, run in ROM

- Configured program, boot from ROM, run in RAM

- Configured program, boot from ROM, run in ROM

The memory maps for each of these are summarised below.

Single processor, boot from ROM, run in RAM

The memory map for this case will have the same layout as the single processor, boot from link programs.

Single processor, boot from ROM, run in ROM

It is not known at collect time where in memory the ROM is to be placed. There-fore, the start and end addresses of the code segments are given as offsets from the start of ROM, and are annotated as such. Items such as **workspace** will have absolute addresses allocated, if the program is targetted at a specific processor type.

An example for this case is given in figure 12.7. The example was produced by the following command line:

```
icollect -t -ro -rs 8k simple.lku -p simple.map       (UNIX)
icollect /t /ro /rs 8k simple.lku /p simple.map (MS-DOS/VMS)
```

where: **simple.lku** was produced by compiling and linking the example pro-gram **simple.occ** for a T414 in the default halt on error mode.

```
icollect : INMOS toolset collector
Sun Version 2.0.25

Memory map for processor 0 (Booting and running in ROM)T414
Load Start is 80000188,  HALT ON ERROR
  LOW priority process 'Init.system'
    Code from 'sysproc.lib', file offset 6901
      ROM OFFSET                            #B00       #D20      544
      ROM entry offset                      #B02
    Workspace                          #80000708  #80000778     112

  HIGH priority process 'System.process.b'
    Code from 'sysproc.lib', file offset 29562
      ROM OFFSET                            #AA4       #B00       92
      ROM entry offset                      #AA5
    Workspace                          #800006D4  #800006EC      24

  LOW priority USER process
    Code from 'simple.lku', file offset 2
      ROM OFFSET                            #4D0       #AA4     1492
      ROM entry offset                      #4D0
    Workspace                          #80000188  #800002B0     296
    Vectorspace                        #800002D4  #80000354     512

Parameter data                         #80000580  #800006D4     340
```

Figure 12.7 Memory map file for a single processor program run in ROM

Configured program, boot from ROM, run in RAM

The layout of the memory map for this case will be the same as that for the boot from link configured program.

Configured program, boot from ROM, run in ROM

For this case the root processor will shown in the same format as the single processor case, run in ROM. Some memory locations being expressed as offsets from the beginning of ROM.

For the other processors in the network will appear as the boot from link case.

12.6   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.

Type	Data	Unit
`INT32`	Interface descriptor size	bytes
`[]BYTE`	Interface descriptor	–
`INT32`	Compiler id size	bytes
`[]BYTE`	Compiler id	–
`INT32`	Target processor type	–
`INT32`	Version number	–
`INT32`	Program scalar workspace requirement	words
`INT32`	Program vector workspace requirement	words
`INT32`	Static size	words
`INT32`	Program entry point offset	bytes
`INT32`	Program code size	bytes
`[]BYTE`	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 oc it will always be 10 which indicates no static parameter is present. A value of 11, indicates the presence of a static data 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.

12.7 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 RAM for code 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.

12.8 Alternative bootstrap loaders

If not otherwise specified, **icollect** uses the standard INMOS bootstraps. The correct code for the application program is chosen from a library of bootstraps compiled for different error modes.

The collector can be directed to use other bootstrap loader programs 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 F 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.

12.9 Use of the icollect 'Y' option

The collector option '**Y**' has two effects on the program being built:

- It disables interactive debugging of the program.

- It reduces the amount of memory used.

For programs compiled and linked for a specific transputer type, this option will cause **icollect** to produce a program that uses less memory. However, programs compiled and linked for transputer classes '**TA**' or '**TB**' will not build when this option is used.

The effects of disabling interactive debugging are described in section 4.5.

The disabling 'Y' option may only be used in conjunction with the 'T' option and will be ignored if specified for a configured program.

12.10 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.

12.10.1 Warnings

The following messages are prefixed with 'Warning-'. They are only generated when the 'T' option is used (single processor mode).

Extra disable option on command line ignored

The user has configured the program with interactive debugging disabled and has specified the 'Y' option to the collector.

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.

Program configured with interactive debugging enabled, option ignored

The user has configured the program with interactive debugging and has specified the 'Y' option to the collector. This 'Y' option is ignored and the boot file is built.

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.

12.10.2 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.

Cannot patch parameter data for processor class

The user has specified the '**Y**' option with a linked unit for a processor class. The collector cannot initialise some of the data without a linked unit for a specific processor type.

Command line parsing error at *string*

Unrecognised command line option.

Debug file already specified

More than one debug file was 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 `occonf` 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 `occonf` 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 `occonf` used are incompatible.

Illegal processor type

Unrecognised processor type. Either the file has been corrupted or `icollect` and `occonf` 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, or by the configurer using different files to the ones the collector has found.

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 by
> the collector are not the same file.

Linked unit module not found in : *filename*

> The required library module is missing or has been corrupted. This mes-
> sage 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 (with optional K or M suffix) or
> hex. Hex numbers must be introduced by '#' or '$'.

Memory size string too long

> Specified memory size is too large i.e. it is greater than 8 digits.

More than one parameter statements

> The collector expects only one *parameter* statement per processor. Ei-
> ther the file has been corrupted or the versions of `icollect` and
> `occonf` 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 (to the user program) in the input file
> is not present. 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 (to the user program) in the input file is not of the correct form, indicating that the process interface is not one recognised by the collector. (See section 12.3). This message will only appear if the collector is invoked with the 'T' option (single processor mode).

Print map file already specified

More than one print map file was specified. Specify one only.

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 occonf 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 occonf 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 is not equal to that specified to occonf 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

Illegal ROM size specification. ROM size must be given in decimal (with an optional K or M suffix) or as Hex. Hex numbers must be introduced by '#' or '$'.

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 given in decimal (with optional K or M suffix) or hex. Hex numbers must be introduced by '#' or '$'.

Stack size string too long

Specified stack size was too large i.e. greater than 8 digits.

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.

13 `icvlink` — TCOFF convertor

This chapter describes the file format convertor tool `icvlink` which converts object files from *Linker File Format* (LFF) to *Transputer Common Object File Format* (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`.

13.1 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, and FORTRAN compilers and the D705/D605/D505 releases of the occam 2 compiler.

All object files produced by the latest INMOS Toolsets are generated in a format known as *Transputer Common Object File Format* (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 tools supplied with the current toolsets.

The conversion to TCOFF may take place at different stages in the development process depending on the user's requirements. Figures 13.1 to 13.3 illustrate three different approaches to using `icvlink`. Notice that in all three approaches the conversion is performed before the configuration stage.

In figure 13.1, 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 13.2.2.

Figure 13.2 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 13.3 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 shaded symbols, in the figures, represent both i/o files in LFF format and previous issues of particular tools. **Note**: where t*xx* has been used it would be equally valid to use **.bin** (see section 13.2 below).
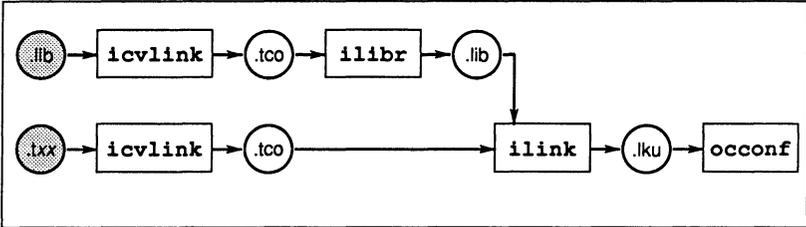

Figure 13.1 Converting compilation and library modules
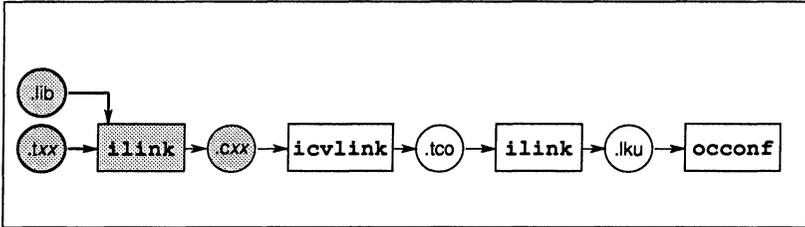

Figure 13.2 Converting linked object module
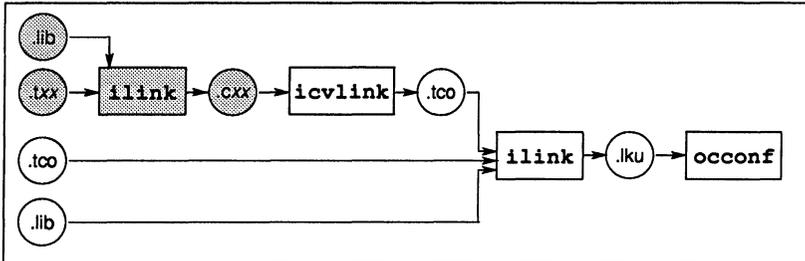
Figure 13.3 Conversion followed by linking with new code

When occam or C 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.

For occam programs the calling conventions for arrays of channels have changed for the new toolset. This will cause problems if you attempt to pass (as a parameter) a channel array from a module compiled with the current toolset compiler to a converted module or vice versa. The convertor will warn the user of any arrays of channels found in a module and will report which routines they are found in. It will also embed a warning message in the actual module, which will be displayed during the linking process.

13.2 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.
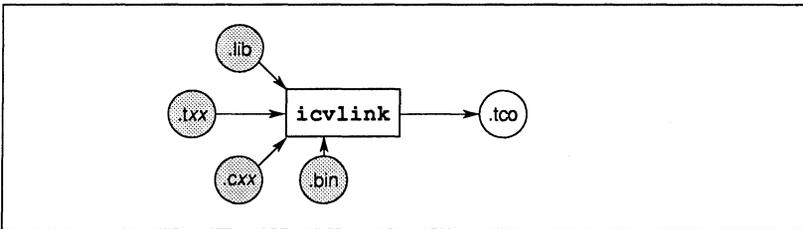
Note: The file extensions of the input files, pertain to default file extensions used by previous issues of INMOS toolsets (e.g. the IMS D705/D605/D505 and IMS D511A/D611A/D711D products), where:

.lib is the extension of a library file.

.txx is the extension of a compiled occam file.

.cxx 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.

options is a list of options given in table 13.1.

Options must be preceded by '−' for UNIX based toolsets and '/' for VMS and MS-DOS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

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.

Options must be preceded by '−' for UNIX based toolsets and '/' for VMS and MS-DOS based toolsets.

Options may be entered in upper or lower case and can be given in any order.

Options must be separated by spaces.

Any string not recognised as an option is treated as the name of the file to be converted.

Table 13.1 `icvlink` command line options

Examples

```
icvlink myprog.t4x
```

In this example `icvlink` is used to convert an occam object file which has been compiled for a T4 transputer. The output filename will default to `myprog.tco`.

```
icvlink myprogc.bin
```

In this example `icvlink` is used to convert an object file, produced by the IN-MOS 3L Parallel C compiler. The output filename will default to `myprogc.tco`.

13.2.1 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.

13.2.2 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.

13.2.3 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.

13.3 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, (see section 4.3).

- Transputer class TC is converted to transputer class T5.

For more information on transputer classes and error modes see sections 4.3 and 4.4.

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.

13.4 Summary of rules for using `icvlink`

1 When occam or C 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. Although LFF and TCOFF libraries may use the same standard names, the format of TCOFF libraries and the calling conventions used, are completely different to LFF library conventions.

13.5   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 section 2.12.1.

13.5.1   Warning Messages

filename - symbol, **implementation of channel arrays has changed**

Channel arrays are now represented differently. This should be remembered when mixing code compiled with different generations of the occam compiler or configurer.

13.5.2   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 found/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.

14 `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.

Chapter 7 describes how to debug occam transputer programs.

14.1 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.

14.1.1 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.

14.1.2 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. Breakpoint mode debugging requires the use of two or more transputers.

Certain symbolic functions and Monitor page commands are only available in breakpoint mode.

### 14.1.3 Mixed language debugging

When debugging programs constructed from a mixture of languages from different INMOS toolsets (occam and C for example), you should always use the version of **idebug** with the highest version number (as displayed in the Help or Monitor page). This is true for all versions of **idebug** with a version number greater than **V2.00.00**. This will ensure that no toolset incompatibilities occur (for instance **idebug** supplied with the first release of the ANSI C toolset does not fully understand the output from **occonf**).

14.2 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 (preferably two).

The relationship of the root transputer to the host computer and the rest of the network is illustrated below.
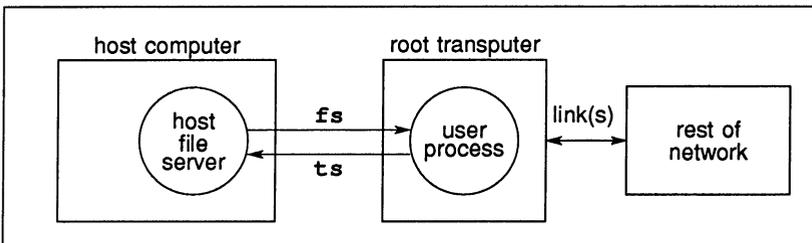
Two procedures are used to debug programs in post-mortem mode, depending on whether the application is configured to use the root transputer. Programs that use the root transputer are referred to in this chapter as **R-mode** programs, and programs that do not use the root transputer are referred to 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.

14.2.1   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 14.3 gives a summary of the command sequences to use for the two program modes on different board types.

14.2.2   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 15 and 24 respectively.

14.2.3   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.

14.2.4   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.

14.2.5   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. Because memory locations etc. are simulated, this
option only requires the root transputer in order to execute the debugger (even
when used with a bootable file for a network of transputers).

This option may also be used to explore the features of the debugger.

14.2.6   Methods for breakpoint debugging

Breakpoint mode debugging does not require use of the memory dump tool be-
cause 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.

14.3 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 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 on the command line.
>
> Options must be separated by spaces.

idebug is unique amongst toolset tools in that when invoked with command line options its driver program does not automatically reset (or analyse as appropriate) the root transputer. This is due to the diversity of hardware configurations where the appropriate sequence may not be obvious to the driver. Because of this, the task of selecting the appropriate **iserver** command is delegated to the user.

Failure to supply the appropriate **iserver** reset (**sr**) or analyse (**sa**) options along with **idebug** command line options will result in **iserver** failing to boot **idebug**.

Only when invoked with no command line options at all will **idebug** automatically reset the root transputer and display its own help page.

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 14.1 Debugger command line options

14.3.1   Toolset file types read by the debugger

The debugger uses information within files produced by toolset tools in order to establish the hierarchy of components used to produce a bootable file. The different types of files within the toolset are described in section 2.9.

Table 14.2 provides a list of file types used by the debugger (in roughly the same order the debugger interrogates them):

File extension	Description
`.btl`	Bootable to be debugged.
`.cfb`	Configuration data file.
`.clu`	Configuration object file.
`.lku`	Linked unit generated by linker.
`.tco`	Object file generated by compiler.
`.lib`	Library file.
`.occ`	occam source code file.
`.inc`	occam include file.
`.pgm`	occam configuration file.
`.c`	C source code file.
`.h`	C include file.
`.dmp`	Debugger dump file.

Table 14.2 File types read by debugger

With the exception of a dump file which must have a `.dmp` filename extension, the debugger will accept different extensions for a particular file type. (For example the extensions used by `imakef` such as `.tah` which can be used instead of `.tco`).

14.3.2   Environment variables

`idebug` requires three environment variables to be set up on the host system (in addition to those required to build a bootable file):

ITERM   Defines key mappings for debugger symbolic functions and
                         some Monitor page commands.

IDEBUGSIZE   Defines the amount of memory available on the root trans-
                         puter board. This variable must be specified for idebug to
                         work correctly (idebug requires at least 1Mbytes of avail-
                         able 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.

14.3.3 Program termination

If the program terminates on issuing the server terminate command 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]`

14.3.4 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* (See section 14.4.1).
For the commands to use on other board types see section 14.4.

```
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  linknumber  -sa
idebug  bootablefile  /m  linknumber  /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 '**n**' 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. The same action may be achieved when using the debugger in breakpoint mode with a subsystem wired *subs* (see section 14.4.1) by use of

the Monitor page 'Y' option (see section 14.6).

Symbolic functions and Monitor page commands that support breakpointing are absent in the post-mortem debugger.

14.3.5 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 performed by `idump`.

14.3.6 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 14.4.

> `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 through-out 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 *iq* 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 the dummy code onto the processors clears each 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.

```
PROC dummy ()
  SKIP
:
```

So that the iserver is terminated correctly the root processor should execute the following variation:

```
#INCLUDE "hostio.inc"
#USE     "hostio.lib"

PROC root.dummy (CHAN OF SP fs, ts)
  so.exit (fs, ts, sps.success)
:
```

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, configure and 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.

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.

14.3.7   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 part 2, appendix G.

Key-mapped symbolic functions and Monitor page commands are listed in section 14.6.5.

14.4   Debugging programs on INMOS boards

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. (See section 6.4.1).

14.4.1 Subsystem wiring

On transputer boards the subsystem signal is 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 these subsystem signals are propagated to all transputers on the network, whereas on B004 boards the signals are not propagated at all.

14.4.2   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 14.3 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 to 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 6.

14.4.3   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.

14.5   Debugging programs on non-INMOS boards

If your hardware 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 break-point 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.

14.6   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 occam toolset release.

Board	Wiring	Mode	Command line(s) to use
TRAM	*down*	T	idebug *program* –b *linknumber* –sr –set –st
			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 –set –st
			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.

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 14.4.3.

Table 14.3 Commands to use when debugging B004 and TRAM boards

14.6.1 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.

14.6.2 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 MOSTNEG INT to the hexadecimal value using modulo arithmetic. This is useful when specifying transputer addresses which are signed and start at MOSTNEG INT. For example, on a 32 bit transputer %70 is interpreted as #80000070 and on a 16 bit transputer as #8070.

14.6.3 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.

14.6.4 Editing keys

The following string editing functions are available for on-screen editing of strings for certain commands:

Key	Effect
[START OF LINE]	Move the cursor to the beginning of the string.
[END OF LINE]	Move the cursor to the end of the string.
[DELETE LINE]	Delete the string.
[←]	Move the cursor left one character.
[→]	Move the cursor right one character.
[↑]	Replace the current string with the string used in the previous invocation of the command.
[DELETE]	Delete the character to the left of the cursor.
[RETURN]	Enter the string.

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.

⇑ will not be applicable to some commands.

14.6.5 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.

14.6.6   Summary of main commands

Key	Meaning	Description
A	ASCII	View a region of memory in ASCII.
B ‡	Breakpoint	Display the Breakpoint menu enabling break-points 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

§ = String editing functions available, see section 14.6.4.

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 post-mortem debug session.
?	Help	Display help information.
‡ = Breakpoint mode only		

14.6.7   Symbolic-type commands and scroll keys

Key	Description
$\boxed{\text{TOP}}$ ♯	Locate to the last instruction executed on the current processor.
$\boxed{\text{RETRACE}}$ ♯	Switch to symbolic mode and perform symbolic operation.
$\boxed{\text{RELOCATE}}$ ♯	Switch to symbolic mode and perform symbolic operation.
$\boxed{\text{HELP}}$ ♯	Display help information.
$\boxed{\text{REFRESH}}$ ♯	Re-draw the screen.
$\boxed{\text{LINE UP}}$ ♯ $\boxed{\text{LINE DOWN}}$ ♯ $\boxed{\text{PAGE UP}}$ ♯ $\boxed{\text{PAGE DOWN}}$ ♯ $\boxed{\uparrow}$ $\boxed{\downarrow}$	Scroll the currently displayed memory, disassembly, or queue.
$\boxed{\leftarrow}$ $\boxed{\rightarrow}$	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 for-
mat, 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 or-
dered from left to right in each row. Unprintable characters are substi-
tuted 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.

☐ Note that `main ()` relates to the fixed C function called at C program startup. `entries` (entrypoints) relate to the first procedure called at occam program startup for non-configured programs and the start of configuration code for configured programs.

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 set breakpoints at the entrypoint of a process (at configuration level).

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.

☐ **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 oc-currence 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. The command interprets all the memory as 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 enables a program source file to be displayed within the symbolic debugging environment for a particular processor. This 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 command 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.

The behaviour of this command will differ depending on whether `icconf`, the configurer supplied with the ANSI C toolset has been used rather than using `occonf` the occam configurer supplied with this toolset, or indeed no configurer at all (in the case of programs running on a single processor). For example `icconf` may be used to combine occam modules with mixed language modules during configuration.

The differences in the behaviour of the command are described below:

Behaviour of command when `occonf` is used

The debugger first prompts for the filename of a *linked* object module. The full linked filename (including extension) must be supplied.

        Linked unit filename ?

The linked filename must be specified because the debugger needs to know which linked unit incorporated by a configurer #USE directive you are interested in.

The debugger then prompts for the filename of a *compiled* object module contained within the selected linked unit. The full object filename (including extension) must be supplied.

        Object module filename ?

The object module filename must be specified because the debugger extracts the source code filename from the debug information in the compiled object file.

Note: Editing keys may be used with this command to provide a simple history mechanism (see section 14.6.4).

☐ At each prompt this command may be aborted by pressing ⌐RETURN⌐ with no typed input.

Behaviour of command when no configurer or `icconf` is used

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: Editing keys may be used with this command to provide a simple history mechanism (see section 14.6.4).

At each prompt this 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 de-
sired 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.

ASCII arrays are displayed in the format used by the Monitor page
command 'ASCII'. Other types are displayed both in their normal rep-
resentation 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**. (For details of these functions see part 2, section 1.10).

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:
```

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.

Note: Editing keys may be used with this command when starting the program, see section 14.6.4.

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 debugger checks in configured programs that the link the root processor has been booted from matches that expected by the configurer. It it does not, the following message is displayed:

```
Booted from link N < Should be link M !!!  >
```

The format of the display is similar to the following:

```
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, stack (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 total memory usage. Total memory usage indicates the amount of memory used by a user program; this may include a region of memory at the beginning of freespace. This will occur for configuration code which is reclaimed from freespace before execution of a user program starts. (The configuration code resides in a region of memory which may be safely overwritten by the action of user code because the configuration code finishes executing before the user code starts executing).

Total memory usage is the minimum memory size (for a particular processor) you need to specify to the configurer, collector or place in `IBOARDSIZE` as appropriate.

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 dis-
play using the CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN,
PAGE UP, and PAGE DOWN keys.

Note: In breakpoint mode this command may show the details of a
process more than once. The following string '< ! >' next to the queue
heading serves a reminder that this may occur.

Processes which belong to the debugging kernel are also displayed
and identified with the string '(Runtime kernel)'.

☐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 pro-
      cesses 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 activ-
      ity 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.

      Note: This command is of limited use when used in conjunction with
      the occam configurer **occonf**. This is because user processes can-
      not have names assigned to them in the same manner as with the C
      configurer **icconf**.

☐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.

☒   Exit

This command returns to symbolic mode and locates to the current address.

☒   Enter post-mortem debugging

This command allows the debugger to be switched into post-mortem mode from breakpoint 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. It has the same effect as re-invoking the debugger with the command line '**M**' option.

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 will only work if the subsystem is wired *subs* (see section 14.4.1). For a subsystem wired *down*, you will need to quit and restart the debugger using the Monitor page '**M**' command line option (instead of the previous breakpoint command line '**B**' option).

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.

14.6.8 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. |

14.7 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 14.4. 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 14.4 Debugger symbolic functions

| INSPECT | This function allows you to find the type and value of any occam symbol. To inspect a symbol, use the cursor keys to position the cursor on the required symbol and press INSPECT.

If the cursor is not on an occam symbol when you press INSPECT, you are requested to specify a symbol name. Type ENTER to abort the INSPECT operation, or type a name followed by ENTER. Spaces and the case of the letters in the name are significant. If the symbol is an array, elements from the array can be selected using constant integer subscripts enclosed in square brackets ('[' and ']'). If no subscripts were supplied, you are prompted to supply them.

The symbol is checked that it is in scope with the line to which the debugger last located. This may not be the same as the current cursor position. If the symbol is not in scope at that location, or not found at all, one of the following messages is displayed:

Name 'symbol' not in dynamic scope

Name 'symbol' not found

Information displayed

If the name is in scope, its type and value are displayed, together with its address in memory. If it is an array, and subscripts were supplied, its type, value, and address are displayed. If it is a short **BYTE** array, it is displayed in ASCII. If it is any other type of array, its dimensions are displayed. If it is a channel, and is not empty, the **Iptr** and **Wdesc** of the process waiting for communication, and its priority, are displayed. If it is a **PROC** or **FUNCTION** name, its entry address, and nested workspace and vectorspace requirements are displayed (no address is displayed for library names). For protocol names and tags, timers, and ports, only types are displayed.

If there is too much information to be displayed on one line, it is displayed in two parts. The symbol's name and type is displayed first, then after a short pause, its value and address.

Inspecting arrays

The debugger displays the size and type of the array, and prompts for subscript values. For example:

[5] [4] INT ARRAY 'a', Subscripts ?

Press ENTER to obtain the address of the array, or enter the required subscripts, which must be in the correct range.

The subscripts should be typed either as decimal constant integer values, or as integers separated by commas, for example '[3][2]', or '3, 2'. Spaces are ignored.

To simplify access to values such as 'a[i]' you may type 'a[!]'; the '!' character is replaced by the value of the last integer displayed.

Scrolling arrays

Instead of supplying subscripts for an array element, the debugger allows you to scroll through the elements of an array while inspecting in symbolic mode. It also allows you to see a short 'segment' of a BYTE array; you can move this segment up and down like a window into the array.

When asked for a subscript, you may add '++' onto the end (or '++' on its own; this assumes a subscript of zero). Then instead of displaying only that element of the array, the debugger also displays the following message on the second line of the screen:

```
Press [UP] or [DOWN] to scroll, any
other key to exit :
```

You may use the ↑ and ↓, cursor keys to scroll through the elements of the array. The debugger will not allow you to scroll past the beginning or end of the array. Pressing any other key will return you to normal symbolic mode.

BYTE arrays have another useful feature. If you add a single '+' to the subscripts, the debugger displays a 'segment' of 16 bytes starting at those subscripts. You may again scroll through the array by using the ↑ and ↓, cursor keys. As before, you cannot scroll past the beginning or end of the array.

If you use a single '+' on a non-BYTE array, it behaves exactly like '++'.

☐   **Inspecting memory**

To inspect the contents of any location in memory, specify an
address rather than a symbol name. Type the address as a
decimal number, a hexadecimal number (preceded by '#'), or the
special short form %h...h, which assumes the prefix #8000...
The debugger displays the contents of the word of memory at
that address, in both decimal and hexadecimal.

For more versatile displays of memory contents, use the func-
tions available at the 'Monitor page' (see section 14.6).

Inspecting placed channels

For channel variables that have been placed into a specific mem-
ory location the INSPECT function displays both the address of
the location and its value.

Note: that this is a change from previous versions of `idebug`
where only the address was displayed.

Channels can be examined in detail using the CHANNEL function.

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 topmost procedure, no backtrace is possible.

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 non-occam vari-

This function displays Hex values of non-occam variables as well as their decimal values. The default is to display integral types in decimal format only.

This function does not apply to occam variables.

INTERRUPT

This function forces the debugger to enter the Monitor page without stopping the program when breakpoint debugging.

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.

Note : A side effect of this command is that the debugger suspends **iserver** communications in order to preserve debugger output to the screen.

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.

14.7.1 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.

This function is commonly used to continue a process which has stopped on a program error, see 7.11.

| MODIFY | This function changes the value of a variable in transputer memory. Use this function as you would | INSPECT | to select a variable for modifying (press | MODIFY | and specify the name of the variable).

Specifying an empty string aborts the | MODIFY | operation.

Once a variable is selected the debugger prompts for a new value. The new value should be specified in the expected occam type (as specified within the prompt) although there are a few relaxations to this rule to allow for implicit casts when using the debugger (see below). **REAL32** and **REAL64** values must be given in the correct occam format.

The following occam types may be freely mixed to provide implicit type casts so long as the value is defined within the destination type:

    BOOL   BYTE   INT   INT16   INT32   INT64

The following are examples of valid modification values:

Destination variable type	Modify value
REAL32	42.0
INT64	TRUE
INT	'a'
BOOL	'*#00'
INT16	#A0
INT32	$1A
BYTE	42

The following are examples of invalid modification values:

Destination variable type	Modify value
REAL32	42
INT64	2.0
BOOL	'*#02'
INT16	32768
BYTE	-1
BYTE	#100

14.8 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.

14.8.1 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.

14.8.2 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 *iq* 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.

14.8.3 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 *iq* 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* **(lptr: #***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. (e.g. `ilist` *filename*).

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 *iq* 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.

Compiler complains that any of the following debug support functions are not found:

DEBUG.ASSERT
DEBUG.STOP
DEBUG.MESSAGE
DEBUG.TIMER

You have omitted the `#USE "debug.lib"` directive required to incorporate the debug support functions.

Configuration info inconsistent with linked unit

You have probably relinked a component of a program and forgotten to reconfigure it.

Configured for post-mortem debugging only

You have explicitly disabled interactive debugging (either via configurer or collector options).

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 14.8.1 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*"

The meaning of this error message depends on which configurer you have used:

`occonf`

You have configured your program with the configurer 'RE' option to enable memory layout re-ordering.

`icconf`

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.

File has changed since configuration "*filename*"

You should rebuild the program again.

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 14.8.1 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 (as defined by the debugging kernel). This is caused by a channel that has become corrupted (normally by overwriting the location of the channel). You should ensure that no compiler checks have been disabled to prevent accidental corruption.

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.

Name *symbol* is not in dynamic scope

The symbol *symbol* exists in the module, but is not in scope from where the debugger last located to. In order to inspect the symbol you must locate to a new position where the symbol is in scope.

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 *iq* systems products. These products are available separately
from your local INMOS distributor.)

(Probe Go) : Processor *number* - Incorrect 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. (See section 7.18.12).

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 14.8.1 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 $\boxed{\text{EXIT FILE}}$ 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 declared at configuration level (*number*)
Too many processes used at configuration level (*number*)

The debugger requires more memory in order to operate on this many processes (see section 14.8.1 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 14.8.1 on how to overcome this).

Unable to read iterm environment variable

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 spec-
ified.

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 (to configuration code)

This normally occurs when you try to backtrace from the program's top-
most procedure into the bootstrap routine which is not supported sym-
bolically by the debugger (i.e. the configuration code area).

You cannot backtrace from here (to Iptr: #nnn, Wdesc: #mmm)

An attempt to backtrace from a procedure or function has failed because
the resultant process details are invalid (e.g. `Iptr` is not in the Code
area).

The `Iptr` and `Wdesc` shown are those of the invalid process which
supposedly called the current procedure or function.

If you suspect that this is not the case you should use $\boxed{\text{INFO}}$ before
backtracing to check that the current process details are valid. (They are
normally only invalid when incorrect process details have been specified
with the Monitor page 'O' command). Corruption of the stack (workspace)
is another possible cause; you should ensure that no compiler checks
have been disabled to prevent accidental corruption.

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 a transputer type (instead of a class)

The program you are trying to debug is for a transputer class (either TA or TB); the debugger needs to know the actual processor type (e.g. T425).

You should retry using the debugger with the command line 'C' option to specify the appropriate processor type.

You must specify the application boardsize in IBOARDSIZE to be $<=$ #10000

On a T2 the maximum memory size is 64K (#10000).

15 `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.

15.1   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.

15.2   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.

15.2.1 Example of use

Assuming an `IBOARDSIZE` of 100000:

```
idump core 100000
```

15.3   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.

16 `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.

16.1 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*.

16.2   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 16.1.

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 16.1 **iemit** command line options

Note: that if option 'E' is selected i.e. interactive mode, then <u>no</u> other options may be specified on the command line.

The operation of **iemit** in terms of standard file extensions is shown below:
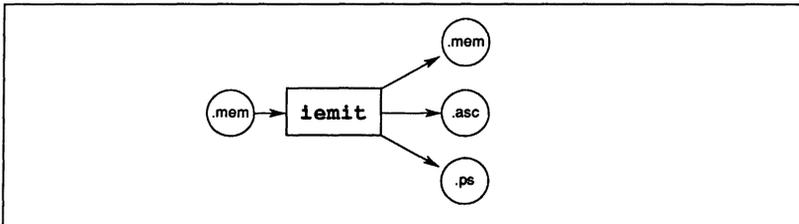
Examples of use

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**

Note: `iemit` will make use of the ITERM host environment variable, if it is available, otherwise it will use defaults.

16.3   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 16.7.

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.

16.4   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.

16.4.1   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 16.2 and an example of the display page is given in figure 16.1.

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.

16.4.2   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 16.2.

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 16.3) 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 16.4, 16.5 and 16.6.

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 16.5 and 16.6).

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 16.2 `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 16.3 `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 **not-MemS0** 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 16.4 **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 16.5 `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 16.6 **iemit** page 1 parameters

16.4.3 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 16.7 lists the timing information displayed on this page while an example of the display is given in figure 16.3.

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 16.7 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.

16.4.4 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 16.8 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 16.8 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 16.4.

16.4.5 Page 4

This page shows graphically the timing for a memory read cycle. An example of the display page is given in figure 16.5.

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.

16.4.6 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 16.6.

16.4.7 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 16.7.

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.

16.5 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 16.1 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 16.2 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
TAVOL   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 16.3 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 16.4 Example **iemit** display page 3

```
Page 4    |  1  |  2  |  3  |  4  |  5  |  6  |

ProcClock   / ‾‾\___/ ‾‾\___/ ‾‾\___/ ‾‾\___/

notMemS0 0)= ‾‾‾_____/

notMemS1 1)= ‾‾‾\_

notMemS2 2)= ‾‾‾‾‾‾_____/

notMemS3 3)= ‾‾‾‾‾‾‾‾‾‾‾‾_____/

notMemS4 4)= ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

MemWait     ‾‾‾_____/
READ CYCLE
MemAD       ‾‾‾‾‾‾‾‾>------------------<    >-<
                    `Read data latched here ^

notMemRd (r)= ‾‾‾‾‾‾‾‾‾‾‾‾_____/
```
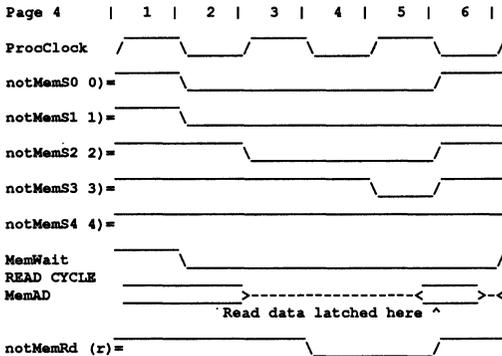
Figure 16.5 Example **iemit** display page 4
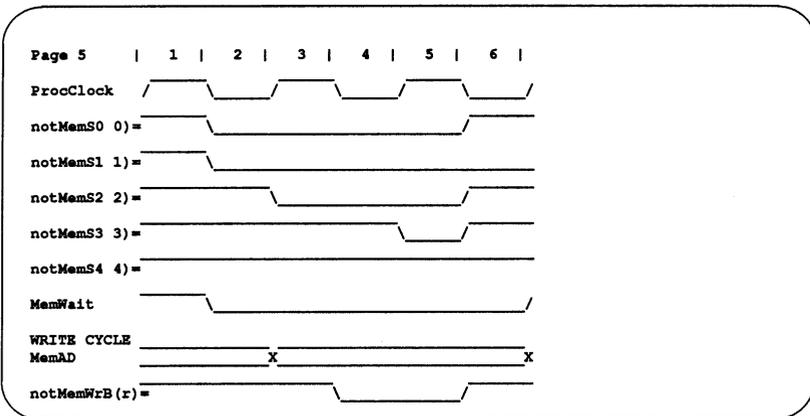
Figure 16.6 Example *iemit* display page 5
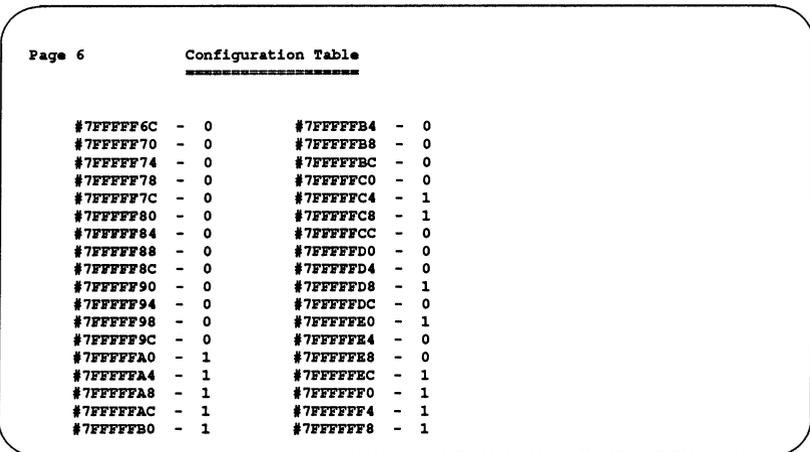
Figure 16.7 Example *iemit* display page 6

16.6   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.

16.7   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 16.2.

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 16.8.

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 16.9. Further information about specifying parameters is given in section 16.4.2.

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 16.9 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 16.10 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
```

16.8   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.

16.9   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 the IMS D705/D605/D505 toolsets.

options is a list of one or more options from table 16.11.

> 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 16.11 `icvemit` options

The operation of `icvemit` in terms of standard file extensions is shown below:
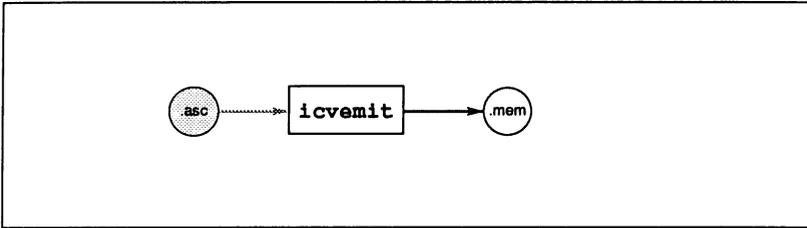
Note: the file extension of the input file pertains to previous issues of the toolset.

Example

```
icvemit memconfig.asc -i -o memconfig.mem          UNIX
icvemit memconfig.asc /i /o memconfig.mem    (MS-DOS/VMS)
```

16.10 `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.

17 `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.

17.1 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 17.1 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 16.

17.2 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 `occonf` and `icollect` tools. See chapters 26 and 12 respectively.
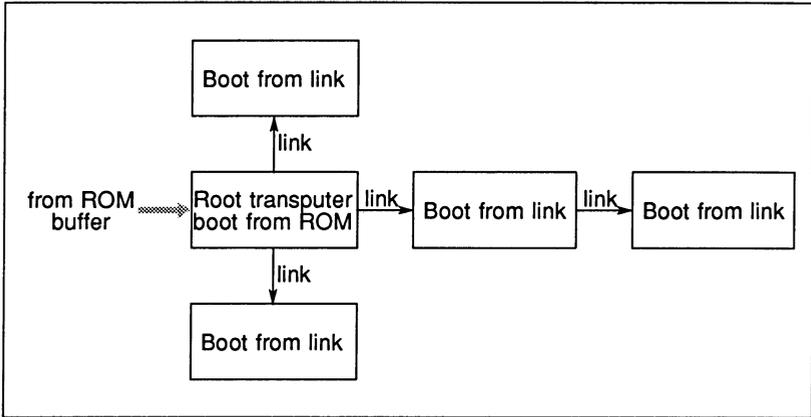
Figure 17.1 Loading a network from ROM

17.3 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.
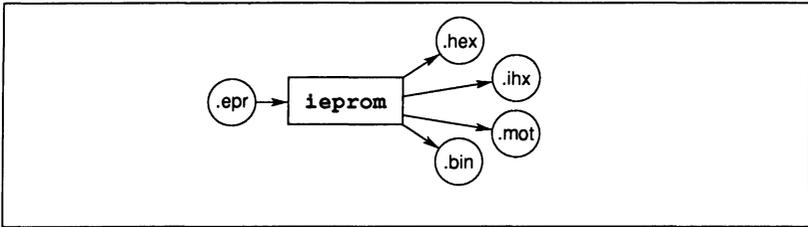
17.3.1 Examples of use

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)

17.4 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 17.2). 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 17.1 and 17.2.

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 16 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 17.1 `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 17.2 `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 ...
```

17.5 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 17.2.
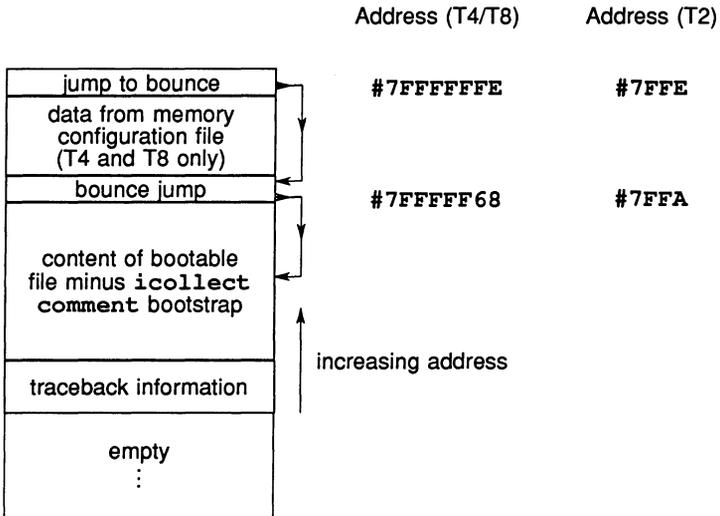
Figure 17.2 Layout of code in EPROM

17.5.1 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.

17.5.2 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.

17.5.3 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.

17.5.4 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.

17.6 **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 17.7.

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.

17.6.1 .Binary output

This file is in binary format and simply contains all bytes output. There is no additional information such as address or checksums.

17.6.2 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.

17.6.3 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.

17.6.4 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.

17.6.5 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.

17.7 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 17.2).

17.7.1 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).

17.7.2 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.

17.7.3 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.

17.8 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
```

17.9   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.

18 `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, and ends with some hints about how to build efficient libraries from separate modules.

18.1 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:

- Library files already generated by `ilibr`.

- Object code files produced by `oc` the occam 2 compiler.

- Object code files produced by `icc` the ANSI C compiler.

- Linked object files (see section 18.2.3).

- Object code files produced by the convertor tool `icvlink`.

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.

18.2 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 object files (excluding library 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.
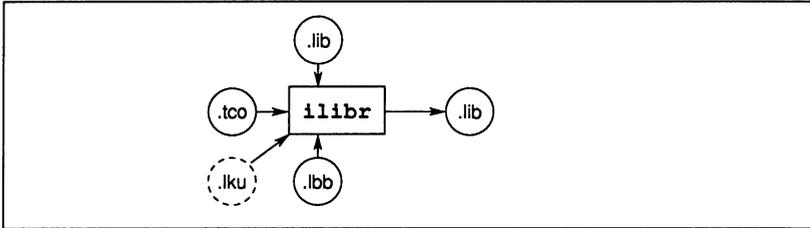
To invoke the librarian use the following command line:

▶ **ilibr** *[filenames]* *{options}*

where: *filenames* is a list of input files separated by spaces.

options is a list of one or more options, in any order, from table 18.1.

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 brief help page is displayed.

Example

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 18.1 `ilibr` command line options

18.2.1 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.

18.2.2 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.

18.2.3 Linked object Input files

The librarian will accept linked object files as input. The facility to create libraries of linked modules provides an easy method of specifying input to the configurer **occonf** (see chapter 5). Such library files should only be referenced from a configuration description.

The librarian will generate an error if an attempt is made to include both linked units and compiled modules in a single library. In addition, libraries of linked object modules must *not* be used as input to the linker **ilink**. This is because the linker does not accept linked units as input files.

18.3   Library modules

Libraries are made up of one or more selectively loadable modules. A module is the smallest unit of a library that can be loaded separately. Modules are selected via the library index.

18.3.1 Selective loading

Libraries can contain the same routines compiled for different transputer types and (if non-C code is used) in different error modes.

For occam modules the compiler identifies which modules need to be selected according to requirements of the main program. The linker **ilink** then makes this selection. For example, if the program is compiled for an IMS T414 only modules compiled for this processor type or for processors in a compatible transputer class are loaded. The linker selects library modules for linking on the basis of usage. Only those modules that are actually used by the program are linked into the program.

For C modules it is the linker that decides which library modules are used. The linker will select the library modules best suited to the compilation units.

18.3.2 How the librarian sorts the library index

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.

The librarian compares and sorts modules according to a number of factors including attributes set by the compiler options used. These determine for example, the instruction set of the module and influence run-time execution times.

For example, where two library modules were derived from the same source but compiled for classes **TA** and **T8**, the librarian would place the **T8** module first because it uses a larger instruction set. Modules compiled with the '**Y**' option will be placed before copies which do not have the '**Y**' attribute. This is because the '**Y**' attribute causes the code to execute faster. The librarian orders the index entries such that the first valid entry is always the 'best choice'. If two entries are found to be identical the librarian will issue a warning.

18.4   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.

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 are text files. They may be created for a specific library by invoking the **imakef** tool 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.

18.5   Building libraries

This section describes the rules that govern the construction of libraries and contains some hints for building and optimising libraries.

18.5.1 Rules for constructing libraries

1 Routines of the same name in a library must be compiled for different transputer types or 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 two error modes.

4 Libraries that contain modules with interactive debugging enabled are treated as though they also contain a copy with interactive debugging disabled. (When interactive debugging is enabled, channel input/output is performed via library calls otherwise transputer instructions are used).

18.5.2 General 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 and with different input/output access to channels. 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.

18.5.3 Optimising libraries

This section suggests how the user might tailor a library for its intended use.

It is possible for the user to optimise the size and content of any libraries which he builds himself, to target appropriate processors, improve the speed of code execution and to provide the best code for a given processor.

For occam libraries there are three aspects of the library build that may be optimised, these are: error mode, method of channel input/output and transputer type. While identifying how the library is intended to be used, the user should consider the following:

- Whether the library is to be targetted at one or two specific processors or a wide range of processors. The transputer type specified for the compilation of a library module determines the instruction set used. 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 targetted at fewer transputers than classes TA and TB.

- For floating point operations, classes T5 and TB provide better code and therefore better execution times than class TA.

- The error mode that the library is compiled for will affect the size of the library. As stated above a library created from modules compiled in UNIVERSAL mode will behave as if it contains a copy of the code for both HALT and STOP mode.

- On the current range of transputers, code compiled in HALT mode will tend to execute faster than if it is compiled in STOP or UNIVERSAL error modes.

- The method of channel input/output will affect both the availability of the interactive debugging facility and the speed at which the code will be executed.

  When interactive debugging is enabled, channel input/output will be implemented via library calls. When interactive debugging is disabled using the compiler 'Y' option, transputer instructions are used for channel input/output. This leads to faster execution times. However, disabling interactive debugging for one module of a program, will disable this facility for the whole program.

- The final consideration may be whether the versatility of the library should be reduced in order to create a smaller library.

Outlined below are three different approaches to optimisation. The third approach provides the greatest level of flexibility in its application. The experienced user may refine these guidelines to his specific requirements.

For a detailed description of transputer types and error modes, see sections 4.3 and 4.4.

Library build targetted at specific transputer types

This method of building a library will limit the use of the library modules to specific
transputer types and error modes. It is recommended as the simplest strategy
to use when the following options are known for each module:

• Target transputer type.

• Error mode, (i.e. HALT, STOP or UNIVERSAL).

• Method of channel input/output.

All modules to be included in the library are compiled for the same error mode
and method of channel input/output and each module must be compiled for each
target transputer type. The resulting library may be large and contain a certain
amount of duplication.

For example, if the following options are used:

T414 and T425 processor types, HALT error mode and channel input/output via
library calls, each module should be compiled for the following:

Processor type	Error mode	Method of channel I/O
T414	HALT	Via library calls i.e. interactive debugging enabled.
T425	HALT	Via library calls.

Semi-optimised library build targetted 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:

Processor class	Error mode	Method of channel I/O
T2	UNIVERSAL	Via library calls.
TA	UNIVERSAL	Via library calls.
T8	UNIVERSAL	Via library calls.

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 targetted at all transputer types

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 classes TA and T8. This will provide modules which can be run on all 32-bit transputers.

2 If any of the modules perform floating point operations, compile these modules for class TB as well.

For 16-bit transputers it should be sufficient to compile all modules for class T2.

18.6 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 2.12.1.

18.6.1   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.

18.6.2   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.

19 `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 toolset linker, goes on to describe the command line syntax and command input files, and ends by describing some of the linker options and other features of the linker's operation.

19.1   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 `oc` the occam 2 compiler, the librarian `ilibr`, or by `icvlink` the file format convertor. 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 *Transputer Common Object File Format* (TCOFF) supported by this release of the toolset. However, the linker can be directed to produce output files in *Linker File Format* (LFF). In this format the output is compatible with either the `iboot` or `iconf` tools used by previous (IMS D705/D605/D505) releases of the toolset.

The operation of the linker in terms of standard input and output file extensions is shown below.
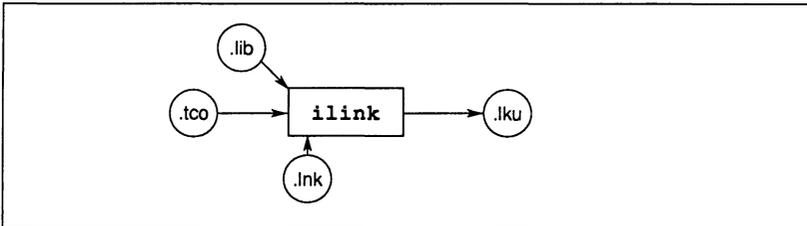
19.2   Running the linker

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 19.1 and 19.2.

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.

Example:

UNIX based toolsets:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
icollect simple.lku -t
iserver -se -sb simple.btl

MS-DOS/VMS based toolsets:

oc simple
ilink simple.tco hostio.lib /f occama.lnk
icollect simple.lku /t
iserver /se /sb simple.btl

In this example a compiled file is linked for the default transputer **T414**, using
the linker indirect file **occama.lnk**. This 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**.

Table 19.1 **ilink** command line options

Option	Description
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 19.4.2 below.
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** and **iconf** tools (supported by previous (IMS D705/D605/D505) releases of the toolset).
LC	Specifies that the output is to be generated in LFF format, for use with the **iconf** tool (supported by previous (IMS D705/D605/D505) releases of the toolset).
EX	Allows the extraction of modules without linking them.
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 a 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 modules.

Table 19.2 **ilink** command line options

19.2.1 Default command line

A set of default command line options can be defined for the tool using the **ILINKARG** environment variable. Options must be specified using the syntax required by the command line.

19.3 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.

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.

19.3.1 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: 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'. Should a procedure called 'proc_a' be 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).

Note this directive is not applicable to occam.

#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. 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. If there is more than one such symbol the linker will warn that there is an ambiguity.

#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** directive enables the user to define the order in which particular modules occur in the executable code.

In order to use this directive one or more occam modules must have been compiled using the compiler directive **#PRAGMA LINKAGE**. Details of the syntax may be found in section 25.

The compiler directive associates a section name with the code of a compilation module. A section name may take the default value "**text%base**" or "**pri%text%base**" or a name specified by the user.

The linker will place modules, that are 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 any modules identified by **second%section%name**, followed by any other modules.

19.3.2 Linker indirect files – supplied with the toolset

Three linker indirect files are supplied with the toolset. Each file contains a list of library files which may be required to be linked but which are additional to those obviously referenced by the program. These include compiler libraries and support for interactive debugging. Each file supports different target processor types, as shown below.

Linker indirect file	Target processors
`occam2.lnk`	T212/T222/T225/M212
`occama.lnk`	T400/T414/T425/TA/TB
`occam8.lnk`	T800/T801/T805

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.

19.4   Linker options

19.4.1 Processor types

A number of options are provided to enable the user to specify the target processor for the linked object file, see table 19.1. 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 4.3 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.

19.4.2 Error modes – options H, S and X

Three error modes are provided by the toolset for compiling and linking programs:

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 19.3 indicates the compilation error modes which are compatible and the possible error modes they may be linked in.

Compatible compilation error modes	ilink options
HALT, UNIVERSAL	H
STOP, UNIVERSAL	S
UNIVERSAL	X

Table 19.3 `ilink` error modes

Note: modules which have been compiled in UNIVERSAL error mode may also be linked in this mode. However, if the resulting linked unit is then processed by the `icollect` tool it will be treated by `icollect` 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.

19.4.3 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 `occonf` 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 the IMS D705/D605/D505 releases of the toolset. The '**LB**' option produces a file compatible with the **iboot** and **iconf** tools used by previous toolsets. The specifed 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.

19.4.4 Extraction of library modules – option **EX**

This option allows the extraction of modules unlinked. The linker functions as normal except that it does not produce a linked unit as output. The output produced by this option is a concatenation of the component modules that would have made up the linked unit. By default the output file produced will have the extension **.lku**, although it is not a linked unit. An alternative output filename and extension can be specified using the **ilink** 'O' option.

This mechanism can be used for creating sub units for linking at a later date or for extraction of modules from libraries. The linker 'U' option and the **#REFERENCE** directive are particularly useful for controlling the content of the output.

19.4.5 Display information – option **I**

This option enables the display of linkage information as the link operation proceeds.

19.4.6 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 ($>= 1 Kbytes$) 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 at link-time.

19.4.7 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 a symbol name as its argument, 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. If there is more than one such symbol the linker will warn that there is an ambiguity.

19.4.8 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.

A link map file is a text file containing information such as the name of the main entry point and the position of modules in the code file. For more information about link map files see section 19.6.

19.4.9 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.

19.4.10 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.

19.4.11 Disable interactive debugging – option Y

This option applies to occam modules only.

The linker supports interactive debugging by default. When interactive debugging is enabled the linker will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. Interactive debugging must be enabled in order to use the interactive features of the debugger.

Interactive debugging may be disabled for occam modules by using the linker 'Y' option. This option forces the linker to select modules that use sequences of transputer instructions for input and output, resulting in faster code execution.

19.5   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.

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.

19.6   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).

Figure 19.1 provides an example of a link map file produced when linking the program `simple.occ` in default mode.

```
ilink module information file version 2.00.03
Target: simple.lku T414 H
Main entry point: simple from simple.tco, simple.occ
Workspace: 75 words Vectorspace: 128 words
PROC simple(CHAN OF SP fs,
CHAN OF SP ts,
[]INT memory)
   SEQ
     fs?
     ts!
  :
Section text%base              1816 bytes
Total Code                     1816 bytes

File          Source      Mode        Addr    Size    Reference
simple.tco    simple.occ  T414 H         0     164    -
hostio.lib    echoline.occ  TA H       164     152    so.read.echo.line
hostio.lib    exit.pah    TA H         316     100    so.exit
hostio.lib    getkey.occ  TA H         416      96    sp.getkey
hostio.lib    spwrite.occ TA H         512     188    sp.write
hostio.lib    wstring.occ TA H         700     452    so.write.string
hostio.lib    puts.occ    TA H        1152     180    sp.puts
hostio.lib    sowrite.occ TA H        1332      96    so.write
virtual.lib   virtual.tmp TA X        1428     264    VIRTUAL.IN%
debug.lib     semprocs.tmp TA X       1692     124    SEMAPHORE.WAIT%
```

Figure 19.1 Example link map file produced for `simple.occ`

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.

19.7   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 21 *must* be used.

19.8   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 section 2.12.1.

19.8.1   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**

> 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.

filename1 - **usage of** *symbol* **out of step with** *filename2*

> Languages such as occam 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.

19.8.2   Errors

filename - **name clash with** *symbol* **from** *filename*

> In languages such as occam entrypoints may be scoped, ie. 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 such scoping has been mixed with a scoped language such as occam 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*

> Languages such as occam 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 implimentation 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 - 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.

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.

Invalid or missing descriptor for main entry point *symbol*

The specified main entry point to the program does not have a valid occam descriptor. This occurs if the wrong symbol name has been used, for example a data symbol in C.

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 have been supplied 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 have been supplied to a directive.

Unknown error modes 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 existance since the LFF format was defined.

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 existance since the LFF format was defined.

filename - line number - **unrecognised directive** *directive*

> An unrecognised directive has been found in an indirect file.

19.8.3 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*

20 `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`.

20.1 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 occam 2 compiler, the ANSI C compiler, the linker, librarian, file convertor, configurer and collector tools. Text files, file formats produced by `ieprom` and separately compiled units designed to be dynamically loaded via `KERNEL.RUN` can also be displayed using `ilist`. Files in editable ASCII format are listed without further processing.

The `ilist` tool will also list compilation and linked units in Linker File Format (LFF). (This file format was used by previous versions of INMOS toolsets e.g. the IMS D705/D605/D505 and IMS D511A/D611A/D711D series 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.

20.2 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:

- *Procedural data* – procedural interfaces in the form of occam function or procedure headings for all entry points in each module, showing parameters, data types and channel usage. This data can only be provided for code produced by oc the occam 2 compiler.

- *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.

- *Index data* – the content of library indexes.

20.2.1 Example displays used in this chapter

The example displays used in this chapter show the output from the lister when used to list the following files:

- **simple.occ**

- **simple.tco**

- **simple.lku**

- **simple.cfb**

- **hostio.lib**

The example program **simple.occ** (introduced in chapter 4) was processed by the occam 2 toolset to produce a compilation file, linked unit and bootable file.

simple.occ was compiled for a T414 processor in HALT mode using oc. The compiled code was then linked with **hostio.lib** and the linker indirect file **occama.lnk**.

The occam library **hostio.lib** is used to demonstrate certain options, for example, the display of the library index.

20.3 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 20.1. 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 command syntax.

ilist will attempt to identify the file type by its contents. If filenames only are supplied, **ilist** uses the default option **w**.

Example:

    ilist simple.tco -a   UNIX based toolsets
    ilist simple.tco /a   MS-DOS and VMS based toolsets

In this example **ilist** is being instructed to display all the symbol data for the file **simple.tco**.

Table 20.2 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.

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. Alternatively the **ilist** 'o' command line option may be used to redirect output to a specified file.

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 lister onto a 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 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 20.1 **ilist** command line options

20.3.1 Default command line

A set of default command line options can be defined for the tool using the **ILISTARG** environmental variable. Options must be specified using the syntax required by the command line.

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 20.2 Recommended options

20.4   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.

20.5   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 is extensive and detailed and 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 plus the number of its origin.

- Module name.

- Target processor.

- Error mode.

- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

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:

Symbol attribute	Description
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. In the example in figure 20.1 four origin symbols are listed.

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 20.1 provides an example of the symbol data displayed for a single module simple.tco by the A option.

```
text&base                   R-X-- -E------ 0    simple.occ        T414 H
local&text                  ----- L------- 1    simple.occ        T414 H
simple.occ:5166ACD2         ----- -E-----O 2    simple.occ        T414 H
so.write.string             ----- --I----- 3    simple.occ        T414 H
wstring.occ:5D454245        ----- --I----O 4    simple.occ        T414 H
so.read.echo.line           ----- --I----- 5    simple.occ        T414 H
echoline.occ:5E442F81       ----- --I----O 6    simple.occ        T414 H
so.write.nl                 ----- --I----- 7    simple.occ        T414 H
so.write.string.nl          ----- --I----- 8    simple.occ        T414 H
so.exit                     ----- --I----- 9    simple.occ        T414 H
exit.pax:AB2F3979           ----- --I----O 10   simple.occ        T414 H
simple                      ----- -E------ 11   simple.occ        T414 H
simple'ws                   ----- -E---U-- 12   simple.occ        T414 H
simple'vs                   ----- -E---U-- 13   simple.occ        T414 H
```

Figure 20.1 Example output produced by the **A** option.

20.6   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 output from this option gives an ASCII dump, in hexadecimal format, of the code for each module. It can be used on any object code.

When used to display object code produced by the occam compiler, the code for 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 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 it is replaced by a dot (.).

Figure 20.2 shows the code listing produced by listing **simple.tco** with the C option.

Note: the display normally appears embedded within the display produced by the **T** option. Only the lines corresponding to the code listing are illustrated here.

```
000010E6 000060B8 274421FB D77C23F4 D640D57D    ..'.'D!..|#..@.}
000010F6 D12147D0 244721FB 7A792696 7DD315D2    .!G.$G!.zy&.}...
00001106 76D17BD0 147A7925 9F7D7A79 26907DD1    v.{..zy%.}zy&.}.
00001116 46D02440 21FB7A79 24987641 25F27641    F.$@!.zy$.vA%.vA
00001126 25F240F0 21F3F474 F021F37D D174D07B    %.@.!..t.!.}.t.{
00001136 7A792490 7DD07730 7A79239E B822F050    zy$.}.w0zy#..".P
00001146 6C656173 65207479 70652079 6F757220    lease type your
00001156 6E616D65 203A4865 6C6C6F20 FFC99A3B    name :Hello ...;
00001166 00002020 20202020 20202020 20202020    ..
00001176 20202020 20202020 20202020 20202020
```

Figure 20.2 Example output produced by the C option.

20.7 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. For C programs the option will also display 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.

- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

Figure 20.3 shows a small section of the output produced by listing the occam 2 library **hostio.lib** using the E option.

```
so.read.echo.int64      -> reint64.occ      OCCAM    TA   X
so.read.echo.hex.int64  -> rehint64.occ     OCCAM    TA   X
so.read.echo.real32     -> rereal32.occ     OCCAM    TA   X
so.read.echo.real64     -> rereal64.occ     OCCAM    TB   H
so.read.echo.real64     -> rereal64.occ     OCCAM    TB   S
so.read.echo.real64     -> rereal64.occ     OCCAM    TB   X
so.read.echo.real64     -> rereal64.occ     OCCAM    T800 H
so.read.echo.real64     -> rereal64.occ     OCCAM    T800 S
so.read.echo.real64     -> rereal64.occ     OCCAM    T800 X
```

Figure 20.3 Example output produced by the E option.

20.8 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 in hex 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 20.4 shows the display produced by listing the text file **simple.occ** using the H option.

```
00000000 23494E43 4C554445 2022686F 7374696F      #INCLUDE "hostio
00000010 2E696E63 220A5052 4F432073 696D706C      .inc".PROC simpl
00000020 65202843 48414E20 4F462053 50206673      e (CHAN OF SP fs
00000030 2C207473 2C205B5D 494E5420 6D656D6F      , ts, []INT memo

  .       .       .       .       .         .       .
  .       .       .       .       .         .       .

00000170 66657220 46524F4D 20302046 4F52206C      fer FROM 0 FOR 1
00000180 656E6774 685D290A 20202020 736F2E65      ength]).    so.e
00000190 78697420 2866732C 2074732C 20737073      xit (fs, ts, sps
000001A0 2E737563 63657373 290A3A3A 0A0A          .success).::...
```

Figure 20.4 Example ouput produced by the **H** option.

20.9 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 20.5 shows the output produced by the M option when listing a compiled file **simple.tco** followed by a linked unit **simple.lku**.

In this example the following line indicates where the output for **simple.lku** begins.

        MODULE: LINKED   T414 H

```
MODULE:  OCCAM        T414 H
VERSION: oc simple.occ
COMMENT: occam 2 product compiler (26th January 1990)
COMMENT: occam 2 compiler v1.68 (alpha4.6pre) (17:44:02 Feb 17 1990)
MODULE:  LINKED       T414 H
VERSION: ilink <various>
COMMENT: LINKED FILE
COMMENT: LINKER TCOFF
SC  simple.tco    (0000002) 000000 000159 : -
LIB hostio.lib    (0167103) 000160 000327 : echoline.occ:5E442F81
LIB hostio.lib    (0102336) 000328 000419 : exit.pax:AB2F3979
LIB hostio.lib    (0097204) 000420 000511 : getkey.occ:BC2BD465
LIB hostio.lib    (0151276) 000512 000859 : so.write.nl
LIB hostio.lib    (0088873) 000860 001035 : puts.occ:5AF27B35
LIB hostio.lib    (0086344) 001036 001331 : so.write
LIB virtual.lib   (0001316) 001332 001583 : VIRTUAL.IN
LIB debug.lib     (0005199) 001584 001703 : SEMAPHORE.SIGNAL
```

Figure 20.5 Example display produced by the **M** option.

20.10 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.

- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

Figure 20.6 shows a small section of the output produced by listing the library index for the occam 2 library **hostio.lib**.

```
000156C9  so.gets              OCCAM     TA    X
0001CEA4  so.gets              OCCAM     T212  X
0003DCC9  so.multiplexor       OCCAM     TA    X Y
000447ED  so.multiplexor       OCCAM     T212  X Y
0001A4EC  so.multiplexor       OCCAM     TA    X
00021E18  so.multiplexor       OCCAM     T212  X
000384EE  so.open              OCCAM     TA    X Y
0003F1B2  so.open              OCCAM     T212  X Y
000143DC  so.open              OCCAM     TA    X
0001BB62  so.open              OCCAM     T212  X
```

Figure 20.6 Example display produced by the N option.

20.11 Procedural interface data – option P

This option only applies to occam modules and displays procedural information for all function or procedure headings encountered in the module. 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 20.7 shows the procedural data output for the file simple.tco.

```
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 20.7 Example display produced by the **P** option

20.12 Specify reference – option **R**

This option is used in conjunction with any of the other options to locate a specific symbol within a named library. All library modules that export the symbol are displayed. **Note**: symbol names are case sensitive.

The example in figure 20.8 was generated on a UNIX host, by the following command line:

ilist hostio.lib -e -r so.open

```
so.open              -> open.occ          OCCAM     TA    X Y
sp.open              -> open.occ          OCCAM     TA    X Y
so.open              -> open.occ          OCCAM     T212 X Y
sp.open              -> open.occ          OCCAM     T212 X Y
so.open              -> open.occ          OCCAM     TA    X
sp.open              -> open.occ          OCCAM     TA    X
so.open              -> open.occ          OCCAM     T212 X
sp.open              -> open.occ          OCCAM     T212 X
```

Figure 20.8 Example display produced by the **E** and **R** options.

20.13 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; large amounts of data are produced which may require skilled interpretation.

Figure 20.9 shows part of the full data output for the file **simple.tco**.

```
00000000 LINKABLE
00000002 START_MODULE CORE FMUL FPSUP BIT32 MS=18 H lang: OCCAM ""
00000010 VERSION tool: oc origin: simple.occ
00000020 COMMENT PRINT "occam 2 product compiler (26th January 1990)"
00000051 COMMENT PRINT "occam 2 compiler v1.68 (alpha4.6pre)
                                      (17:44:02 Feb 17 1990)"
00000091 SECTION REA EXE EXP "text%base" id: 0
0000009F SYMBOL LOC "local%text" id: 1
000000AD SET_LOAD_POINT id: 0
000000B0 DEFINE_LABEL id: 1
000000B3 SYMBOL EXP ORI "simple.occ:5166ACD2" id: 2
000000CA DESCRIPTOR id: 2 lang: OCCAM <ORIGIN DESCRIPTOR>
     .         .       .       .       .
     .         .       .       .       .
000010E3 LOAD_TEXT bytes: 160
00001186 ADJUST_POINT -30
0000118B SYMBOL IMP "so.write.string" id: 3
0000119E LOAD_PREFIX size: 6 AP(SV:3-LP) instr: j
000011A7 SYMBOL IMP ORI "wstring.occ:5D454245" id: 4
000011BF SPECIFY_ORIGIN id: 3 origin: 4
000011C3 SYMBOL IMP "so.read.echo.line" id: 5
000011D8 LOAD_PREFIX size: 6 AP(SV:5-LP) instr: j
     .         .       .       .       .
     .         .       .       .       .
00001500 DEFINE_SYMBOL id: 11 SV:1+2
00001508 DEFINE_SYMBOL id: 12 76
0000150D DEFINE_SYMBOL id: 13 128
00001512 DESCRIPTOR id: 11 lang: OCCAM
ws: 76 vs: 128
PROC simple(CHAN OF SP fs,
CHAN OF SP ts,
[]INT memory)
  SEQ
    fs?
    ts!
:
00001568 END_MODULE
```

Figure 20.9 Example display produced by option **T** for a **.tco** file.

The full data listing of a configured file shows how the processes are mapped onto a transputer system and has a different appearance to other displays produced by this option.

Figure 20.10 shows the full data output for the file **simple.cfb**. The example shows that there are three initialisation processes, inserted by the toolset, followed by one user defined process.

```
00000004 OPTION mode: size: 0
00000018 FILE id: 0 index: #D67 origin: #2767F6 name: "sysproc.lib"
0000003B FILE id: 1 index: #26E8 origin: #27ED3C name: "sysproc.lib"
0000005E FILE id: 2 index: #6134 origin: #2778AF name: "sysproc.lib"
00000081 FILE id: 3 index: #2 origin: #27D326 name: "demo.lku"
000000A1 END FILES
000000A9 PROCESSOR  id: 0 type: T414 mode: HALT NOCONF size: -1
       |          name: ""
000000C5 |  PARAMETER offset: 2616 size: 472
000002AD |  PROCESS  id: 0 type: INITSYSTEM mode: LOW_PRI SEQ NO_SEP_STACK
       |  |         name: ""
000002C5 |  |  CODE offset: 2104 size: 496 entry.offset: 2 file.id: 0
000002DD |  |  DATA offset: 0 size: 84 type: STACK param.size: 32
000002F5 |  END PROCESS
000002FD |  PROCESS  id: 1 type: INITOVERLAY mode: LOW_PRI SEQ NO_SEP_STACK
       |  |         name: ""
00000315 |  |  CODE offset: 4816 size: 2924 entry.offset: 1 file.id: 1
0000032D |  |  DATA offset: 4116 size: 672 type: STACK param.size: 28
00000345 |  |  DATA offset: 7740 size: 512 type: VECTOR
0000035D |  END PROCESS
00000365 |  PROCESS  id: 2 type: INIT mode: HIGH_PRI PAR NO_SEP_STACK
       |  |         name: ""
0000037D |  |  CODE offset: 3292 size: 824 entry.offset: 3 file.id: 2
00000395 |  |  DATA offset: 3088 size: 176 type: STACK param.size: 28
000003AD |  END PROCESS
000003B5 |  PROCESS  id: 3 type: USER mode: LOW_PRI PAR NO_SEP_STACK
       |  |         name: ""
000003CD |  |  CODE offset: 344 size: 1760 entry.offset: 2 file.id: 3
000003E5 |  |  DATA offset: 0 size: 308 type: STACK param.size: 36
000003FD |  |  DATA offset: 2104 size: 512 type: VECTOR
00000415 |  END PROCESS
0000041D END PROCESSOR
00000425 END PROCESSORS
0000042D END LINKS
00000435 END CHANNELS
```

Figure 20.10 Example display produced by option **T** for a **.cfb** file.

20.14 File identification – option **W**

This option causes the lister to identify the file type. **ilist** takes a heuristic approach to file identification. The filename is displayed along with the file type. The search path is also given if applicable. This is the default command line invocation if no other option is supplied.

Table 20.3 indicates how the lister classifies file types.

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 20.3 File types recognised by `ilist`

where 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 (IMS D705/D605/D505) of the toolset.

20.15 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.

- Interactive debugging – if disabled indicated by the presence of a 'Y' character. If this field is blank then interactive debugging is enabled.

Figure 20.11 shows a small section of the output produced by listing the occam library **hostio.lib** using the **X** option.

```
REAL64TOSTRING        <- wreal64.occ        OCCAM        T800 S Y
sp.write              <- wreal64.occ        OCCAM        T800 S Y
REAL64TOSTRING        <- wreal64.occ        OCCAM        T800 X Y
sp.write              <- wreal64.occ        OCCAM        T800 X Y
sp.getkey             <- readline.occ       OCCAM        TA    X Y
sp.getkey             <- echoline.occ       OCCAM        TA    X Y
sp.write              <- echoline.occ       OCCAM        TA    X Y
sp.time               <- times.occ          OCCAM        TA    X Y
sp.open               <- opentemp.occ       OCCAM        T212 X Y
sp.close              <- opentemp.occ       OCCAM        T212 X Y
```

Figure 20.11 Example display produced by the **X** option.

20.16 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 section 2.12.1.

20.16.1 Warning messages

filename - reason

> The named file does not conform to a recognised INMOS file format or has been corrupted.

20.16.2 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 found/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.

21 `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.

21.1 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 inter-dependencies 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 with the toolset so that it can be modified for use with other MAKE programs.

21.2 How **imakef** works

imakef operates by working back from the target file to determine its dependencies 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.

21.2.1 Target files

The following table lists the types of object files for which **imakef** can create Makefiles. The file extensions required by **imakef** are also given for each of the files (see section 21.2.2).

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

21.2.2 File extensions for use with **imakef**

imakef identifies files and file types by a special set of file extensions which specify the transputer type and error mode and allow it to produce Makefiles for mixed module combinations. Note that these extensions differ from the standard toolset defaults and must be used for all stages of program development for all types of target file.

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 21.1 in relation to toolset program development.
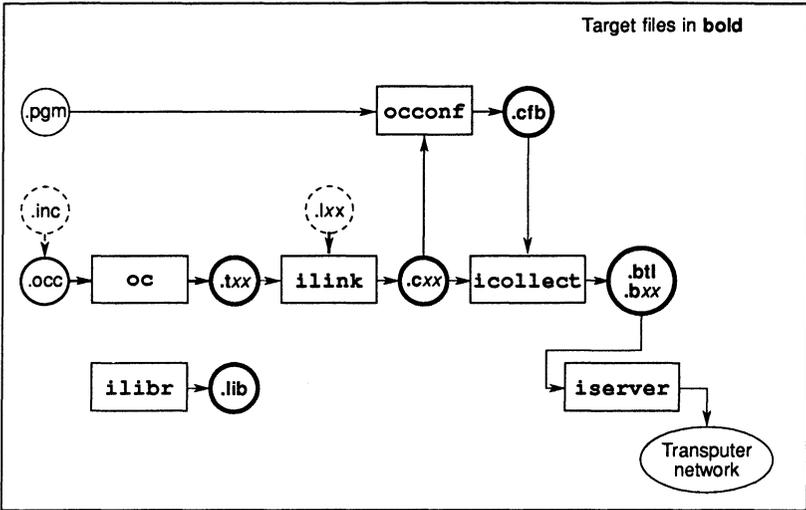
Figure 21.1 Main target files showing file extensions required

Values that can be taken by the second character of certain object files to designate a transputer target 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

Object code may be generated in one of three error modes: HALT, STOP or UNIVERSAL. These are represented by the letters 'h', 's' and 'x' respectively, which may be given as the third character of the file extension.

Examples:

.**t4x** refers to a compiled module targetted for the T4 transputer class, in UNI-VERSAL error mode.

.**t8h** refers to a module targetted for the T800 transputer in HALT error mode.

21.3   Running the Makefile generator

The **imakef** program takes as input a list of files generated by tools in the toolset and generates a Makefile, containing full instructions of how to build the application program. The output file is named after the first target filename and is given a .**mak** extension (if no output file is specified on the command line).

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 21.1.

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.

21.3.1 Example of use

        imakef simple.b4h

This creates the Makefile **simple.mak** which when used as input to **MAKE** generates the bootable file **simple.b4h**.

Option	Description
C	This option is only used when incorporating C modules into the program. It specifies that the list of files to be linked is to be read from a linker indirect file.
D	Disables the generation of debugging information in compilations. 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.
NI	Do not include dependencies on ISEARCH.
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 all compilations. The default is to compile with full breakpoint debugging information.

Table 21.1 `imakef` options

21.3.2 Incorporating C modules

`imakef` can also be used for mixed language programs which incorporate C modules by specifying the 'C' option. This option directs `imakef` to look for one or more linker indirect files from which to determine file dependencies.

Linker indirect files must be supplied for all linked units, which include modules supplied in C, on which `imakef` is to be used. Such files take the name of the linked unit to which they relate but have the extension .lnk. The linker indirect file *must* include *all* components of a program, including occam as well as C modules, and any libraries that are used.

The linker indirect file defines the components to be linked and provides a starting point for determining the dependencies of the C components of the program.

An example is given in section 21.4 of how `imakef` may be used to build a mixed language program.

21.3.3 Configuration description files

When `imakef` is required to build a Makefile for a multitransputer program it will look for the presence of a configuration description file which has the same name as the program to be built.

Provided the 'C' option is not used `imakef` will look first for a file with the extension .pgm, (which can be read by the occam configurer occonf). If, however, a .pgm file is not present `imakef` will then look for a .cfs file (which can be read by the ANSI C configurer icconf).

When the 'C' option is used the reverse is true. `imakef` looks first for a .cfs file and if the appropriate file is not present, then it will look for a .pgm file.

21.3.4 Disabling debug data

Two options disable the creation of debug data.

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.

21.3.5 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 or clean as the target.

21.3.6 Files found on ISEARCH

When `imakef` runs, it includes all dependencies in the set of rules. When building a program, it is often not necessary to record dependencies on standard system components such as hostio.lib, which are found using ISEARCH. The 'NI' option prevents `imakef` from recording any dependencies on files found using ISEARCH, thus creating a clearer and more portable Makefile.

21.4 `imakef` examples

This section contains three 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. The third example is for a mixed language program run on a single transputer.

The program used for the first two examples is the pipeline sorter program which is supplied in the examples directory.

21.4.1 Single transputer program

The example program is made up of four files:

```
sorthdr.inc
element.occ
inout.occ
sorter.occ
```

To create the Makefile for the example program use the command:

```
imakef sorter.b4h
```

Note the use of the `.b4h` 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 generator has built-in knowledge of the file name rules of the toolset. In this example, it knows by examining the file name that the program to be built is for a single T414 processor in HALT mode, and that the source of the main body of the program is in the file `sorter.occ`. It reads the file `sorter.occ` and discovers that it uses a library called `hostio.lib`, the two compilation units `inout` and `element`, and two include files, `sorthdr.inc` and `hostio.inc`. It then reads the sources of the include files and compilation units and finds no more file dependencies.

With this information about source file and their dependencies, `imakef` builds a Makefile called `sorter.mak` containing full instructions on how to build the program and creates a linker indirect file `sorter.l4h` (see section 21.7).

To build the program run the MAKE program on `sorter.mak`. The entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

21.4.2 Multitransputer program

This version of the sorter program is configured to place linked units on four processors. The program is made up of the following files:

```
sorthdr.inc
element.occ
inout.occ
sortb3c.pgm
```

To create the Makefile use the following command:

```
imakef sortb3c.btl
```

The `.btl` extension informs `imakef` that the target is a configured program, to be built from a configuration description file called `sortb3c.pgm`.

The configuration description references two linked units:

```
#USE "inout.c4h"
#USE "element.c4h"
```

Note the use of the `.cxx` form of extension instead of the toolset default extension for linked units `.lku`.

`imakef` reads the `.pgm` file and will produce a file called `sortb3c.mak` containing a MAKE description of the program.

To build the program run the MAKE program on `sortb3c.mak`.

21.4.3 Mixed language program

The following example, uses the mixed language program described in sections 9.2.5 and 9.2.6.

To create the Makefile for the example program use one of the following commands:

```
imakef callc.b4h -c                      (UNIX)
imakef callc.b4h /c                      (MS-DOS/VMS)
```

This command informs `imakef` that we wish to create a bootable program for a single T414 processor in HALT mode. The 'C' option tells `imakef` that the program also includes modules written in C.

The example program is made up of the following files:

> `callc.t4h` – Compiled occam module.
> `csubfunc.t4x` – Compiled C module.

`imakef` needs to know the names of the C components of the program, and looks for the associated linker indirect file `callc.lnk`. Because a linker indirect file is supplied to `imakef`, it must contain a list of all modules to be linked.

`callc.lnk` must contain the following files:

> `callc.t4h`
> `csubfunc.t4x`
> `callc.lib`
> `hostio.lib`
> `libcred.lib`
> `#INCLUDE occama.lnk`

The occam module is listed first, since it contains the main entry point of the program. Note the use of the `t4h` and `t4x` extensions. The C module is to be compiled in UNIVERSAL mode, which is the standard mode for the C compiler. UNIVERSAL mode may be called by any of the three occam compilation error modes.

The library `libcred.lib` is the C reduced runtime library used by the C module and `callc.lib` and `hostio.lib` are the occam libraries. `occama.lnk` contains a list of occam compiler libraries which may be required.

With this information `imakef` builds the Makefile `callc.mak`.

21.5   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.

21.5.1   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.

21.5.2 Rules

Rules define the dependencies of object files on other files and specify *action strings* to build those files. For example:

```
example.t4h:  example.occ
$(OCCAM) example -t4 -h -o example.t4h $(OCOPT)
```

This rule first defines the target as the compiled program `example.t4h`, which is dependent on the source file `example.occ` 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.)

21.5.3 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.

21.5.4   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 the output of progress information is to be enabled for all invocations of the compiler, the compiler 'I' option would be added to the `OCOPT` macro which defines the standard combination of options for invoking the compiler. Alternatively a new macro containing only the 'I' option could be defined and added to each compiler action string.

Re-running `imakef`

Once the set of options have been changed in the macros, it is useful to retain this set of options when `imakef` is run again. For this reason, `imakef` will check for the existence of a previous Makefile. If one exists, it will re-use (in the new Makefile) the set of macro definitions from the old one, plus any additional text up to a line marked "`IMAKEF CUT`".

21.6   Library usage files

Library usage files describe the dependencies of a library on other libraries or separately compiled code. They contain a list of files to which the library must be linked before it can be run, and ensure that the correct linker commands are generated. Library usage files are only required when the source of the library is not available.

Library usage files are given the same name as the library to which they relate, but with a `.liu` extension and are created using `imakef`. To create a library usage file, specify the library name and add the `.liu` extension. For example, the following command creates a library usage file for the library `mylib.lib`:

```
imakef mylib.liu
```

When `imakef` is used to create a library usage file no Makefile is generated.

21.7   Linker indirect files

imakef will generate a linker indirect file for occam modules which require linking. The file is named after the target filename but is given an extension of the form **.1**xx. The file consists of a list of modules to be linked, as well as an **#INCLUDE** statement which references a further linker indirect file, containing references to the appropriate compiler libraries. This latter type of linker indirect file is supplied with the toolset, (see section 19.3.2). **imakef** decides which compiler libraries will be required, from the extension of the linked object file to be generated and this determines which linker indirect file is included.

Section 21.3.2 gives details of linker indirect files required when C modules are present. These indirect files have the extension **lnk**.

21.8   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 delimited

The terminating bracket, which determines the options in a library build file, is missing at the given line number.

#SC references are illegal in configuration text

At the given line number in the file there is a **#SC** directive, which is illegal in configuration source code.

#SC, #USE may not reference source files

The directives **#SC** and **#USE** cannot be used to reference occam source code.

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.

22 `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. (Information regarding server programs, supplied with other INMOS products, is given in the Delivery Manual that accompanies this release).

22.1 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 part 2, appendix H.

22.1.1 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 section 21.2.2.

22.2 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 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.

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 22.1 `iserver` options

22.2.1 Examples of use

UNIX based toolsets:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
icollect simple.lku -t
iserver -se -sb simple.btl

MS-DOS/VMS based toolsets:

> *oc simple*
> *ilink simple.tco hostio.lib /f occama.lnk*
> *icollect simple.lku /t*
> **iserver /se /sb simple.btl**

In this example **iserver** is instructed to load the bootable file **simple.btl** and to terminate on error. The example also shows the steps for compiling, linking and booting the program.

22.2.2 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.

It is recommended that all two-letter options beginning with 'S' are reserved.

22.2.3 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 14.3.6.

22.2.4 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'.

22.2.5 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.

22.2.6 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.

22.2.7 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.

22.3 Server functions

This section describes the basic set of server functions. All versions of the *iserver* will support these functions, enabling bootable transputer programs to be used with any version of the toolset.

These functions are not intended for direct use by 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 part 2, appendix H.

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.

22.4   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 requested 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 part 2, appendix H.

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, or the network does not match that defined to the configurer.

23 `isim` — IMS T425 simulator

This chapter describes the T425 simulator tool `isim` that allows programs to be run and tested without **T425** transputer hardware. The chapter explains how to invoke the tool and describes the simulator commands that allow the simulated program to be debugged interactively.

23.1 Introduction

The simulator can run any transputer program that would run on a single IMS T425 mounted on a normal transputer evaluation board and supported by a host running **iserver**. No transputer hardware is required unless you have an MS-DOS host, in which case `isim` does requires a 32-bit transputer processor. This is due to the memory requirements of `isim`.

Because the simulator runs the same code that would be loaded onto a real transputer, any program that runs satisfactorily in the simulator will 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 simulated transputer addresses and to single step the program. The program should be loaded into memory (using the $\boxed{G}$, $\boxed{J}$ or $\boxed{P}$ commands) before breakpoint debugging facilities are used. This ensures that breakpoints are not overwritten during the booting phase.

The simulator can also be used to familiarise new users with transputers and transputer programming and as a teaching aid.

23.2 Running the simulator

To run the simulator use the following command line:

▶   `isim`   *filename* [*programparameters*] {*options*}

where: *filename* is the program bootable file.

> *programparameters* is a list of parameters to the program. The list of
> parameters may follow the `isim` option N and parameters must be sep-
> arated by spaces. See section 23.2.1.

options is a list of `isim` 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.

23.2.1 Passing in parameters to the program

Program parameters can be passed to programs which are simulated on any
host. Parameter passing is equivalent to running a transputer bootable program
using `iserver`.

`isim` will normally parse the command line and any options it recognises as its
own will not be passed to the user program. In cases where options are required
for a user program which clash with one of the `isim` options the 'N' option can
be used. After the 'N' option `isim` ceases parsing the command line for its own
options; the remainder of the command line is simpley passed through to the
user program.

23.2.2 Example of use

`isim simple.btl`

This invokes the simulator on the program named "Simple".

When first invoked the simulator enters the debugging environment. To start the
program invoke the 'G' command. The program runs until it completes success-
fully, 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 commands to examine memory and registers.

Option	Description
B	Batch mode operation. The simulator runs in line mode i.e. full display data is not provided. Commands are read in from the input stream e.g. the keyboard and executed. The commands are not echoed to the output stream e.g. the display screen, as they are executed.
BQ	Batch Quiet mode. The simulator automatically executes the program specified on the command line and then terminates. If an error occurs, the appropriate message will be displayed. The debugging facilities of the simulator are not available in this mode.
BV	Batch Verify mode. Similar to batch mode, except that the commands and prompts displayed when running the simulator in interactive mode are echoed to the output stream e.g. the display.
I	Displays information about the simulator as it runs.
L	Loads the tool onto the transputer board and terminates.
N	No more options for the simulator. Any options entered after this option will be assumed to be program parameters to be passed to the program running on the simulator.
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 23.1 `isim` options

23.2.3 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**.

23.3 Monitor page display

The simulator Monitor page is similar to that of the debugger, which is described in chapter 14. Data displayed at the simulator Monitor page includes the following:

`Iptr`	Contents of instruction pointer (address of the *next* instruction to be executed).
`Wptr`	Contents of workspace pointer.
`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.

If `Wptr` 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).

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.

23.4   Simulator commands

All simulator commands are given at the Monitor page. Many of the commands are similar to those of the debugger Monitor page, however, there are a number of implementation differences. Full descriptions of the commands are given in the following sections.

23.4.1   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 ab-
breviated hexadecimal constants. Hexadecimal constants are specified using the
prefix `#`. Abbreviated hex constants can be created by prefixing the sequence
of hex digits with '`%`' which assumes the hexadecimal prefix '`8000....`'. For
example, the hex number '`8000F8A`' can be specified in the abbreviated form
'`%F8A`'.

23.4.2   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	This option is not supported for the current toolset.
N	Create dump file	Creates a 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 23.4.2.

$\boxed{A}$ – ASCII

This command displays a segment of transputer memory in ASCII format, starting at a specific address. If no address is given the default address **Wptr** is used. Specify a start address after the prompt:

    (Start address (Wptr)) ?

Either press $\boxed{\text{RETURN}}$ 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 blocks of 13 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.

$\boxed{\uparrow}$, $\boxed{\downarrow}$, $\boxed{\text{PAGE UP}}$, $\boxed{\text{PAGE DOWN}}$ keys can be used to scroll the display.

$\boxed{B}$ – Breakpoints

Sets, displays, and cancels break points at specified memory locations or procedure calls. The program should be loaded into memory (using the $\boxed{G}$, $\boxed{J}$ or $\boxed{P}$ commands) before this command is used to set breakpoints. (The $\boxed{D}$ command may also be used prior to this command, to determine where to set breakpoints).

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?

Options are selected by entering one of the single digit commands. The following prompts are displayed depending on the command selection:

command	prompt
1	(break address) ?
3	(break address (ALL))

Pressing $\boxed{\text{RETURN}}$ with no typed input in response to command 1 cancels the option; in response to command 3, it causes *all* breakpoints to be cancelled.

After each breakpoint command the user is returned to the simulator command prompt.

D – Disassemble

The Disassemble command disassembles memory into transputer instructions. Specify an address at which to start disassembly after the prompt:

(Start address (Iptr)) ?

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 thirteen 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.

↑ ↓ PAGE UP PAGE DOWN keys can be used to scroll the display.

G – Go

Starts the program, or continues running the program after a breakpoint or error has been encountered. 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

The Hex command displays memory in hexadecimal. Specify the start address after the prompt:

> `(Start address (Wptr) ?`

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 words of eight hexadecimal digits, with the most significant byte first.

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

The Inspect command can be used to inspect the contents of an entire array.

Specify a start address after the prompt:

> `(Start address (Wptr)) ?`

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    (Not implemented)
7 - REAL32   (Not implemented)
8 - REAL64   (Not implemented)
9 - CHAN

Which occam type ?
```

Give the number corresponding to the type you wish to display or press RETURN to accept the default type. Initially the default will be **HEX**; for subsequent use of the command the default takes the value of the last selected type.

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 thirteen 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** and **CHAN** to the nearest word.

J – Jump into program

Same as G – starts or continues running the program.

L – Links

Displays information about simulated links.

The Links command displays the instruction pointer **Iptr**, workspace descriptor **Wdesc** and priority, of the processes waiting for communication on a link, 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:

```
Link 0 out Iptr: #80000256  Wdesc: #80000091 (Lo)
Link 1 out Empty
Link 2 out Empty
Link 3 out Empty
Link 0 in  Empty
Link 1 in  Empty
Link 2 in  Empty
Link 3 in  Empty

Link 0 connected to Host
Links 1, 2, 3 not connected

Booted from link 0
```

M – Memory map

This option is not applicable to the current version of **isim**, if used the following message will be displayed:

```
Memory Map Invalid
```

N – Create dump file

Creates a 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. The program is loaded into memory but is not automatically run. This command can only be used prior to executing any other instructions.

Q – Quit

Quits the simulator, and returns to the host operating system.

R – Run queue

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 `isim` 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.

S – Single step transputer instruction

This command executes the transputer instruction pointed to by `Iptr`. By repeating the command the user may single step through the program, observing the changes to the process queues and registers, as the display is updated.

T – Timer queue

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.

U – Assign

Assigns a value to a register, `Iptr` or `Wptr`. To assign a value, specify the register by name (abbreviations are permitted), and give a value to be assigned to the register. This enables the program to be re-run (using G or J) with alternative values in the registers.

? – 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.

23.5 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.

23.5.1 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.

23.5.2 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.

23.5.3 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.

23.5.4 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
Wptr           #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
```

23.6   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.

Simulator terminated: Error flag set - *message*

Simulator messages may be output when the simulator halts (i.e. as an error condition).

message can be one of:

> **arithmetic overflow**
> **arithmetic underflow**
> **long oveflow**
> **subscript out of range**
> **count out of range**
> **check single**
> **check word**
> **arithmetic exception**
> **floating point error**

Simulator terminated: *message*

Simulator messages may be output when the simulator halts, due to an invalid operation within the program being simulated.

message can be one of:

attempt made to input from non-existent hard channel

Attempt to input from output link.

attempt made to output to non-existent hard channel

Attempt to output to input link.

attempt to output to unattached hard channel

Attempt to output on unattached link.

attempt to read illegal memory byte at *hhhhhhhh*

> The memory address specified is invalid (not within IBOARD-SIZE).

attempt to read illegal memory word at *hhhhhhhh*

> Invalid memory address or attempt to access non word aligned.

attempt to set illegal memory byte pointer

> Invalid memory address (not within IBOARDSIZE).

attempt to set illegal memory word pointer

> Invalid memory address or attempt to access non word aligned.

attempt to write illegal memory byte at *hhhhhhhh*

> Invalid memory address (not within IBOARDSIZE).

attempt to write illegal memory word at *hhhhhhhh*

> Invalid memory address or attempt to access non word aligned.

high priority process restored from save area

> A swapped out low priority process has been written over during an interrupt.

illegal operand *(nnn)* **to operate command**

> An attempt has been made to execute invalid instruction for the T425.

inputting iserver packet larger than expected

> Illegal ISERVER protocol packet on input.

input from iserver when iserver outputting

> ISERVER packet input before leading output sent.

output iserver packet larger than expected

Illegal ISERVER protocol packet on output.

output to iserver when iserver inputting

ISERVER packet output before response to last output received.

24 `iskip` — skip loader

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.

24.1 Introduction

The skip tool `iskip` prepares a network to load a program over the root transputer by setting up a transparent 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 using `iserver`.

`iskip` can be used to skip any number of processors and load a program into any part of a network, see section 24.2.2.

`iskip` itself may only be executed on 32 bit transputers although it may be used to reach both 16 and 32 bit transputers for target program execution.

24.1.1 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 provides a route-through 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 and memory for the program to be loaded.

This can make debugging transputer programs easier when an extra transputer is available.

24.2 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 24.1.

> 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 24.1 `iskip` options

24.2.1   Skipping a single transputer

This example illustrates how to use **iskip** to skip over the root transputer for the example network shown in figure 24.1.
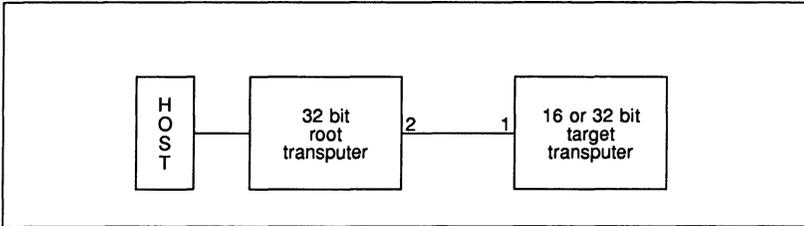
Figure 24.1 Skipping a single transputer

Subsystem wired *down*

```
iskip 2 -r                          (UNIX based toolsets)
iskip 2 /r                          (MS-DOS/VMS based toolsets)
```

In this example **iskip** is invoked for a network where the subsystem is wired *down* (see section 14.4.1). 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.

Subsystem wired *subs*

```
iskip 2 -r -e                       (UNIX based toolsets)
iskip 2 /r /e                       (MS-DOS/VMS based toolsets)
```

In this example **iskip** is invoked for a network where the subsystem is wired *subs* (see section 14.4.1). The '**e**' option has been added to the example, to direct **iskip** to monitor the subsystem error status, see section 24.2.4.

24.2.2   Skipping multiple transputers

This example illustrates how to use **iskip** to skip over two transputers (starting with the root transputer) for the example network shown in figure 24.2

Normally **iskip** is invoked via its driver program; this resets the root transputer and loads the transputer bootable image **iskip.btl** onto the transputer (essentially it performs an **iserver -se -sb iskip.btl** operation).
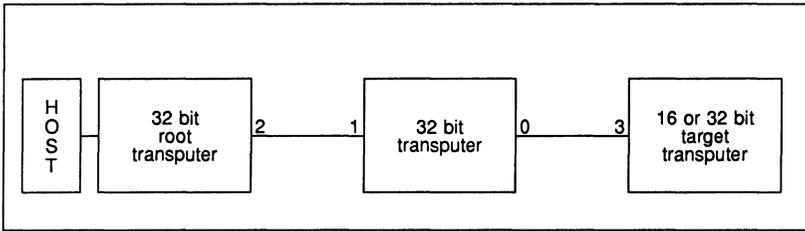
Figure 24.2 Skipping over two transputers

Note: because the root transputer is reset, running `iskip` twice in succession will not achieve any more than running `iskip` once; the second invocation will reset the first and load `iskip` onto the root transputer.

In order to skip over more than one transputer, `iskip` must be loaded onto subsequent transputers by a 'different' method that does not involve resetting the root transputer. This is best illustrated by an example as shown below (for a network wired *subs*):

```
iskip 2 -r -e              (UNIX based toolsets)
iserver -se -ss -sc iskip.btl 0

iskip 2 /r /e              (MS-DOS/VMS based toolsets)
iserver /se /ss /sc iskip.btl 0
```

`iskip.btl` is the transputer bootable component of `iskip`, it may be found in the `itools` directory of this toolset release. For details of toolset directories see the delivery manual supplied with this toolset.

24.2.3 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.

24.2.4 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 6.4).

The 'E' option instructs iskip 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 that 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.

24.2.5 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 network check 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 14.3.6.

24.3   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.

25 oc — occam 2 compiler

This chapter describes the occam 2 compiler oc. It describes the command line syntax and its options, explains about error modes, transputer targets and separately compiled units, and describes the compiler directives in detail. The implementation of channels, which has changed from the previous release of the compiler, in the IMS D705/D605/D505 products, is also discussed. The chapter ends with a description of how usage and alias checking is implemented and a list of error messages.

The compiler implements certain extensions to the language e.g. **RETYPING** channels and channel constructors. These extensions are described in chapter 10 together with facilities for low level programming and dynamic code loading. More detailed information which describes how occam is implemented on the transputer is given in part 2 appendix D. The appendix describes how the compiler allocates memory and gives details of type mapping, hardware dependencies and language.

25.1 Introduction

The toolset compiler implements the occam 2 language targetting to IMS T400, T414, T425, T800, T801, T805 and T2 series transputers. For a full description and formal definition of the occam 2 language see the '*occam 2 Reference Manual*'.

Each compilation of a program must be targetted at a specific transputer type or class, in one of three execution error modes and with interactive debugging either enabled or disabled. The selection or not of interactive debugging determines the method of channel input/output used by the compiler. All components of a program to be run on the same transputer must be compiled for compatible target processors, error modes, and method of channel i/o. The compiler provides comprehensive error message information.

Six directives, extensions to standard occam, are recognised by the occam 2 compiler. These are **#USE, #INCLUDE, #IMPORT, #OPTION, #COMMENT** and **#PRAGMA**. Compiler directives are described in section 25.10.

occam source files can contain references to object code libraries, occam source to be included in the compilation, separately compiled occam code, and code produced by compilers for other languages.

Libraries and separately compiled units must be already compiled before any file which references them can itself be compiled. It is the programmer's responsibility to ensure all components of a program are compiled in the correct order and that object code is kept up to date with changes in the source. This may be assisted by using a MAKE program in conjunction with the `imakef` tool. The `imakef` tool depends on a particular system of file extensions being used. For details of version control using MAKE programs and the `imakef` tool see chapter 21.

The operation of the compiler in terms of standard file extensions is shown below.
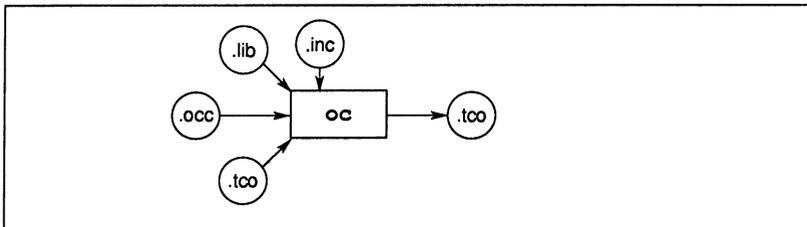
The object file is generated by the compiler in *Transputer Common Object File Format* (**TCOFF**). Object files are required to be in this format to be compatible with other tools in the toolset such as the librarian and linker tools.

25.2   Running the compiler

The occam 2 compiler takes as input an occam source file and compiles it into a binary object file. Command line options determine the target transputer for the compilation, the compilation error mode, and other compiler facilities such as alias and usage checking.

A target processor and compilation error mode should be specified for each compilation. The compiler default is to produce code for the T414 in HALT mode, and for code of this type the transputer target and error mode options may be omitted.

To invoke the compiler use the following command line:

▶ **oc** *filename* *{options}*

where: *filename* is the name of the file containing the source code. If you do not specify a file extension, the extension **.occ** is assumed.

options is a list, in any order, of one or more of the options given in tables 25.1 to 25.3.

> 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 the compilation is unsuccessful, error messages are displayed giving the name of the file and the number of the line where the error occurred. Compiler error messages are listed in section 25.15.

Example:

UNIX based toolsets:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
icollect simple.lku -t
iserver -se -sb simple.btl

MS-DOS/VMS based toolsets:

oc simple
ilink simple.tco hostio.lib /f occama.lnk
icollect simple.lku /t
iserver /se /sb simple.btl

In this example a file is compiled for the default transputer **T414**. This example also shows the steps for linking (using the linker indirect file **occama.lnk**), booting and loading the program.

Option	Description
TA	Compile for transputer class TA (T400, T414, T425, T800, T801, T805)
TB	Compile for transputer class TB (T400, T414, T425)
T212	Compile for a T212 processor.
T222	Compile for a T222 processor. Same as T212.
M212	Compile for a M212 processor. Same as T212.
T2	Same as T212, T222 and M212.
T225	Compile for a T225 processor.
T3	Same as T225.
T400	Compile for a T400 processor. Same as T425.
T414	Compile for T414 processor. This is the default processor type and may be omitted when compiling for a T414 processor.
T4	Same as T414 (default).
T425	Compile for a T425 processor.
T5	Same as T400 and T425
T800	Compile for a T800 processor.
T8	Same as T800.
T801	Compile for a T801 processor. Same as T805
T805	Compile for a T805 processor.
T9	Same as T801 and T805.
H	Produces code in HALT mode. This is the default compilation mode and may be omitted for HALT mode programs.
S	Produces code in STOP mode.
X	Produces code in UNIVERSAL mode.
G	Enables the compiler to recognise the restricted range of transputer instructions, via the ASM and GUY constructs. See part 2 appendix B for the list of permitted instructions.
W	Enables the compiler to recognise the full range of transputer instructions, via the ASM and GUY constructs. See the 'Transputer instruction set: a compiler writer's guide' for the complete list of instructions.
K	Disables run-time range checking. The default is to insert run-time range checks. See section 25.5
U	Disables the insertion of code to perform run-time error checks. The default is to perform run-time error checks. See section 25.5

Table 25.1 occam 2 compiler options

Option	Description
NA	Disables the insertion of run-time checks for calls to **ASSERT**.
L	Loads the compiler and terminates. Useful for loading the compiler onto a transputer system prior to compiling a program.
XM	Compile many programs. The compiler will loop, accepting multiple command lines from the server. Used when the compiler is loaded onto a transputer system.
XO	Compile a single program only. Used when the compiler is loaded onto a transputer system.
A	Prevents the compiler from performing alias checking. This option also disables usage checking. The default is to perform alias checking. When alias checking is enabled, the compiler may insert run-time alias checks. Details of alias and usage checking rules are given in the '*occam 2 Reference Manual*'.
B	Displays messages in brief (single line) format.
C	Disables the generation of object code. The compiler performs syntax, semantic, alias and usage checking only.
D	Generates minimal debugging information. The default is to produce full debugging information. Debugging data is required by the debugger and by the transputer simulator.
E	Disables the use of the compiler libraries. This prevents the compilation of some programs which require 'complicated' arithmetic such as real arithmetic on a processor which does not have a floating point unit. If this option is used and the occam code requires use of the libraries, an error is reported.
I	Displays additional information as the compiler runs. This information includes target and error mode, and information about directives as they are processed. The default is not to display this information.
N	Disables usage checking. The default is to perform usage checking. Usage checking is also disabled by option '**A**'. Details of usage checking rules are given in the '*occam 2 Reference Manual*' and in section 25.13 of this chapter.
O *outputfile*	Specifies the name of the output file. If no output file is specified the compiler uses the current directory and input filename and adds a .tco extension.
R *filename*	Redirects error messages to a file.

Table 25.2 occam 2 compiler options

Option	Description
V	Prevents the compiler from producing code which has a separate vector space requirement. The default is to produce code which uses separate vector space.
Y	Disables interactive debugging with `idebug`. See section 25.7
NWP	Disables warning messages being generated when parameters are not used.
NWU	Disables warning messages being generated when variables or routines are not used.
WD	Provides a warning whenever a name is descoped.
WO	Provides a warning whenever a run-time alias check is generated.

Table 25.3 occam 2 compiler options

25.2.1 Filenames

occam source files can be given any legal filename for the host system you are using. The use of the `.occ` extension for occam source, and the `.inc` extension for files containing declarations of constants and protocols, is recommended.

Output files are specified using the 'O' option. If you do not specify a filename, the input filename is used (minus any directory name) and a `.tco` file extension is added. In this case the file will be placed in the current directory i.e. the directory from which the compiler is invoked.

If you use the Makefile generator tool `imakef` to assist with version control you *must* use the extensions described in chapter 21.

25.3   Transputer targets

The compiler produces code for the IMS T212, M212, T222, T225, T400, T414, T425, T800, T801 and T805 transputers. Command line options are provided to specify the processor type for the compilation. If more than one processor type is specified, the compilation will terminate immediately and an error message will be displayed.

For the purpose of generating common code for several transputer types, transputers are also grouped into the following transputer classes.

Transputer class types	Transputer types
T2	`T212, M212, T222`
T3	`T225`
T4	`T414`
T5	`T400, T425`
T8	`T800`
T9	`T801, T805`
TA	`T400, T414, T425, T800, T801, T805`
TB	`T800, T801, T805`

The concept of transputer classes is to provide the user with greater flexibility in the use of program modules:

• Code can be compiled so that it may be run on a wide range of transputer types.

• Code may be compiled once and then be called by separate programs to run on different processor types.

• Libraries can be created from code compiled for a transputer class which may then be called by programs running on different processor types. The use of transputer classes in libraries enables the size of the library to be kept relatively small without the need for multiple copies for specific transputers.

In order to develop programs which exploit transputer classes a few simple rules must be followed. These rules govern the way modules, which are compiled for different processor types, may call each other and be linked. Full details of transputer types and classes in given in section 4.3. This section explains how to develop programs using single and mixed processor types and definds the rules to be followed.

25.4   Compilation error modes

The compilation error mode determines the behaviour of a program if it fails
during execution. There are two main modes; HALT system and STOP process.
There is also a special mode called UNIVERSAL. Command line options are
provided to select the error mode for the compilation. Specifying more than
one error mode will cause the compilation to terminate immediately and an error
message will be displayed.

The execution behaviour of programs compiled in the different modes is as fol-
lows:

HALT   When an error occurs in the program the transputer halts. This
             is useful for developing and debugging systems and is the de-
             fault mode. For errors to be detected correctly the server must
             be invoked with the 'SE' option.

STOP   When an error occurs the system behaves like the occam
             STOP process, that is the process causing an error does not
             continue. Other processes continue until they become depen-
             dent upon the stopped process. This ensures that a failure
             in one process does not automatically produce failure in other
             processes. Using this mode it is possible to build a system
             with redundancy and enable a system to run even if parts of
             the program fail or processes fail because a time out is ex-
             ceeded.

UNIVERSAL UNIVERSAL mode enables the user to compile code that may
             be run with either HALT or STOP mode in effect. The deci-
             sion about which mode to adopt need not be taken until the
             separately compiled modules are combined into a linked object
             file. On linking the modules, any code that has been compiled
             in UNIVERSAL error mode will adopt the error mode of the
             other modules i.e. either HALT mode or STOP mode. HALT
             and STOP error modes may not be combined on the same
             processor.

Code compiled in either HALT or STOP mode may call code compiled in UNI-
VERSAL mode, however, code compiled in UNIVERSAL mode may only call
code which has also been compiled in UNIVERSAL mode. It cannot call code
which has been compiled in HALT or STOP mode.

All separately compiled units for a single processor must be compiled for com-
patible error modes. Where a library is used the module with the appropriate
error mode will be selected.

Compilation error modes and their effect are described in more detail in 4.4. This section also describes how the command line options 'K' and 'U' may be used to influence the level of run-time checks inserted by the compiler. The section also describes how the UNIVERSAL and UNDEFINED error modes supported by the IMS D705/D605/D505 issues of the toolset may be implemented, using combinations of command line options.

25.5 Enable/Disable Error Detection

By default the compiler inserts code to execute run-time checks for errors it cannot detect at compile time. In some circumstances it may be desirable to omit the run time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes. Three command line options are provided to enable the user to control the degree of run-time error detection performed; they are the 'K', 'U' and 'NA' options.

The compiler option 'K' disables the run-time range checks for the module being compiled. Range checking only includes checks on array subscripting and array lengths.

The compiler option 'U' prevents the compiler from inserting any code to explicitly perform run-time checks. This will disable run-time checks associated with type conversion, shift operations, array access, range validation and replicated constructs such as SEQ, PAR, IF, and ALT.

The 'NA' option prevents the compiler from inserting any code to check calls to ASSERT. In effect, each ASSERT behaves like SKIP. Any calls to ASSERT which can be evaluated at compile time will still be checked.

The effect of using these options is described in detail in section 4.4.1.

25.6   Enabling/disabling warning messages

There are four command line options which allow the user to either enable or disable the generation of certain warning messages produced by the compiler:

- The 'NWP' option disables warning messages being generated when parameters to procedures are not used.

- The NWU option disables warning messages being generated when variables or routines are not used.

- The WD option provides a warning whenever a name is descoped, for example when a name is used twice and one occurrence of it is hidden within an inner procedure. See section 8 of the 'occam 2 Reference Manual' for details of occam scope rules.

- The WO option provides a warning whenever a run-time alias check is generated i.e. to check that variables do not overlap. These checks generate extra code and the user may wish to be alerted to this.

Section 25.15.1 lists the various warning messages which are affected by these options.

25.7   Support for interactive debugging

The occam 2 compiler supports interactive debugging by default. When interactive debugging is enabled the compiler will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. Interactive debugging must be enabled in order to use the interactive features of the debugger.

Interactive debugging may be disabled by using the compiler 'Y' option. This option forces the compiler to use sequences of transputer instructions for channel input and output, resulting in faster code execution.

Code which has interactive debugging disabled may call code which has interactive debugging enabled, but *not* vice versa. However, when interactive debugging is disabled in one part of the program this will prevent the interactive features of the debugger being used on the program as a whole.

25.8   Separately compiled units and libraries

Any group of one or more occam procedures and/or functions may be compiled separately provided they are completely self-contained and make no external references except via their parameters or compiler directives. Separate compilation is used to reduce the need for recompilation, and to split compilations into smaller parts. Separately compiled code is known as a compilation unit.

Any collection of compilation units may be made into a library using the librarian. For details of how to create libraries see chapter 18.

Libraries and compilation units differ in the following ways:

- Libraries are selectively loaded as required by the transputer type and error mode of the compilation, whereas separately compiled units are *always* loaded. If a unit containing incompatible code is used an error is generated, whereas libraries containing incompatible code are ignored.

- Separate compilation units that are contained in libraries can be selectively loaded.

All separate compilation units and libraries must be compiled before the program that references them is itself compiled. An easy way to ensure this is to use the toolset Makefile generator **imakef** with a suitable Make utility. For more details see chapter 21.   .

25.9   ASM and GUY code

Two compiler options are provided to enable the compiler to recognise transputer instructions, via the **ASM** and **GUY** constructs. The 'G' option enables a limited range of instructions, which are listed in part 2 appendix B. The 'W' option enables the full range of instructions. For further details of the low level programming supported by the toolset see chapter 10.

25.10   Compiler directives

The occam compiler supports a number of directives that improve program readability and assist with file referencing. They allow occam source to be included from other files, permit code to be used from separately compiled units and libraries, including C and FORTRAN code, and support the insertion of comments in object code.

Note: That the maximum depth of nesting of include files permitted within a single compilation unit is twenty.

The directives are as follows:

#INCLUDE   — inserts occam source code

#USE   — references separately compiled units and libraries

#IMPORT   — references non-occam compiled code

#COMMENT   — inserts comments in object code

#OPTION   — allows selection of compiler options from within source
               text

#PRAGMA   — references segments of code for mixed language com-
               pilations and/or linking functions

If the compiler 'I' option is used directives are displayed on the screen as the compilation proceeds.

25.10.1 Syntax

Directives must occupy a single line.

Filenames referred to in compiler directives must be enclosed in double quotes ("). Files are located according to the search strategy defined in section 2.10.3.

If double quotes are to be used within a directive, the double quote character must be preceded by an asterisk (*).

The scope of directives are defined, like declarations of constants and protocols, by the level of indentation in the occam source.

When imakef is used, if a filename in a #USE, #INCLUDE or #IMPORT directive does not already have an extension then imakef will add the appropriate extension depending upon the target that it is attempting to build. If you use the Makefile generator tool imakef you *must* use the extensions described in chapter 21.

25.10.2 #INCLUDE directive

The #INCLUDE directive inserts the contents of a named file at the point in the program source where the directive occurs, with the same indentation as the directive.

#INCLUDE files can be used by any number of programs, including separately compiled units, and are commonly used to share common declarations of constants and protocols between several programs.

To track file dependencies within included files use of the `imakef` tool is recommended.

The syntax of the `#INCLUDE` directive is as follows:

   `#INCLUDE` "*filename*" *[comment]*

where: *filename* is the name of the file to be included. The extension must be supplied.

   comment is any text preceded by the characters '`--`'.

The first text after the directive must be the filename enclosed within double quotes ("). All other text on the line is ignored and may be used for comments. Included files may be nested to a depth of twenty levels.

25.10.3 #USE directive

The `#USE` directive allows separately compiled occam units and libraries (in **TCOFF** format) to be referenced from occam source. The file referenced by the `#USE` directive must be compiled for a compatible processor type and compilation mode as the main program, and should be made available in all modes for which the program will be compiled.

The compiler ignores all library modules compiled with a processor type or compilation mode incompatible with the current compilation.

A library may be used in any number of separately compiled units or other libraries, provided that each unit contains the `#USE` directive.

Any names in the library which do not conform to occam syntax, and which have not been translated by means of a **TRANSLATE** pragma will be ignored. **Note**: this means that a **TRANSLATE** pragma must precede its related `#USE` directive. See section 25.10.7.

The syntax of the `#USE` directive is as follows:

   `#USE` "*filename*" *[comment]*

where: *filename* is the name of the object code file. The object file can be a
compiled (.tco) or library (.lib) file. If you omit the file extension, the
compiler adds the extension of the output file. This will be .tco unless
you specified an output filename using the 'O' option.

comment is any text preceded by the characters '--'.

The first text after the #USE directive must be the filename, which must be
enclosed within double quotes ("). All other text on the line is ignored and may
be used for comments.

25.10.4 #IMPORT directive

The #IMPORT directive is retained for compatibility with previous versions of
the toolset (IMS D705/D605/D505 products) which used special interface code
to enable the import of foreign languages. This method may still be used, as
documented in appendix F to the ANSI C toolset user manual.

The #IMPORT directive allows code produced by compatible non-occam com-
pilers to be referenced from occam programs. The code produced must also
be compatible with the toolset linker ilink.

The #IMPORT and #USE directives are in fact the same as far as the compiler
is concerned and either may be used. imakef will also accept either directive,
however, when imakef encounters an #IMPORT directive it assumes that the
module is of another language type and does not look for further dependencies.

The syntax of the #IMPORT directive is as follows:

#IMPORT "*filename*" [*comment*]

where: *filename* is the name of the compiled equivalent occam process. If no
extension is given the .tco extension is assumed.

comment is any text preceded by the characters '--'.

The first text after the #IMPORT directive must be the file name, which must be
enclosed within double quotes ("). All other text on the line is ignored and may
be used for comments.

An example of how to use the #IMPORT directive is given below:

```
#IMPORT "centry.lib"  -- C interface code
        .
        .
        .

PROC.ENTRY(fs, ts, flag, ws1, ws2, in, out)
-- call C language program
```

The parameters supplied in the program call, **flag, ws1, ws2, in,** and **out** are those of the type 2 procedural interface. The program must be linked with C libraries **centry.lib** and **libc.lib.**

The implementation of this method of mixed language programming is described in appendix F of the ANSI C toolset user manual.

Chapter 9 of this manual provides details of an alternative method of mixed language programming where non-occam programs are called directly using library functions.

Changes from the IMS D705/D605/D505 products

The example above, illustrates the method of mixed language programming implemented by the IMS D705/D605/D505 products, with the following modifications to make it compatible with the current toolset:

A library of C entry points is used in place of a prelinked version of the program. This is because all linking now takes place together in one pass.

The C program is referred to by its interface name **PROC.ENTRY**. This means that only one C program may be called from any occam program.

25.10.5 #COMMENT directive

The #COMMENT directive allows comments to be placed in the object code.

The syntax of the #COMMENT directive is as follows:

#COMMENT "*string*"

where: *string* is the text of the comment. Comments must be enclosed in double quotes following the #COMMENT directive. Comments cannot be split over more than one line.

Comments may not appear at the exact position in the object code corresponding with the source code directive, but the sequence of comments in the file is always maintained. Comments are stripped from the object code when it is linked or made bootable.

The main use for the #COMMENT directive is in libraries where it can be used to indicate a version number, record dependencies on other libraries, and hold copyright information.

The binary lister tool ilist can be used to display comments inserted with the #COMMENT directive.

An example of how to use the #COMMENT directive is given below:

```
PROC my.lib ()

  #COMMENT "My library V1.3, 18 May 1988"
  #COMMENT "Copyright me 1988"

  SEQ
    ... library source
```

25.10.6 #OPTION directive

The #OPTION directive allows you to specify compiler options within the source text of a compilation unit. The options apply to the whole compilation and are added to the command line when the compiler is invoked. Only compiler options that relate directly to the source can be specified with the #OPTION directive, namely:

A – disable alias (and usage) checking.

E – disable the compiler libraries.

G – allow sequential code inserts (ASM and GUY constructs).

K – disable the insertion of run-time range checks.

N – disable usage checking.

U – disable the insertion of any run-time error checks.

V – disable separate vector space usage.

W – enable full code inserts, (ASM and GUY constructs).

Y – disable interactive debugging with idebug.

Specifying any other compiler option produces an error.

#OPTION directives can only appear in the file to which they apply; they cannot be nested in an included file. #OPTION directives must also be the first non-blank or non-comment text in the source file. If they are found at any other position in the file an error is reported.

The syntax of the #OPTION directive is as follows:

#OPTION "*optionname {optionname}*" *[comment]*

where: *optionname* is any option permitted in a #OPTION directive. Spaces within the double quotes are ignored. No option prefix character is required in the syntax and none should be specified.

comment is any text preceded by the characters '--'.

The first text after the #OPTION directive must be the list of options enclosed in double quotes. All other text on the line is ignored and may be used for comments.

An example of how to use the #OPTION directive is given below. In the example the unit does not require usage checking but contains transputer code inserts from the restricted set.

```
-- This compilation unit requires sequential
-- code inserts and does not pass the usage check.

#OPTION "G N"

PROC x ()
   ...  body of procedure
:
```

The #OPTION directive should only be used for compiler options that are *always* required.

25.10.7 #PRAGMA directive

The #PRAGMA directive is provided to reference segments of code for mixed language compilations and/or linking functions. The syntax of the #PRAGMA directive is as follows:

> #PRAGMA *pragma-name* {*optional values*} *[comment]*

where: *pragma-name* may take the value:

> EXTERNAL, LINKAGE or TRANSLATE.

> *optional values* may be specified for each type of pragma. The values that the options may take are specific to the pragma being used; they are described below.

> *comment* is any text preceded by the characters '--'. All pragma types may have a comment appended to them.

#PRAGMA EXTERNAL "*declaration*" *comment*

This directive allows access to other language compilations. "*Declaration*" is a PROC or FUNCTION declaration, with formal parameters which correspond to the required calling convention. This is followed (within the string) by two numbers in decimal, indicating the number of workspace slots (words) and optionally the number of vectorspace slots to reserve for that call. The number of vectorspace slots defaults to 0. The number of the workspace slots should not include those needed to set up the parameters for the call. **Note**: that if the vectorspace requirement is zero, then no vectorspace pointer parameter will be passed to the routine.

It is important to ensure that enough space is allocated, both for workspace and vectorspace, because the compiler cannot check for overruns.

The syntax of the declaration is as follows:

formal procedure or function declaration = workspace [, vectorspace]

Examples:

```
#PRAGMA EXTERNAL "PROC p1 (VAL INT x, y) = 20"
#PRAGMA EXTERNAL "PROC p2 (VAL INT x, y) = 20, 100"
#PRAGMA EXTERNAL "INT FUNCTION f1 (VAL INT x, y) = 50"
```

The procedure or function name is the name by which the external routine is accessed from the occam source. It is also the name which will be used by the linker to access the external language function, though this may be modified by use of the TRANSLATE pragma (see chapter 9).

#PRAGMA TRANSLATE *identifier* "*string*" *comment*

This is used to enable linkage with routines whose entry point names do not correspond to occam syntax for identifier names; both imported names to be called by this compilation unit and exported names defined in this compilation unit. An entry point is a name which is visible to the linker. Thus procedures and functions declared at the outermost level of a compilation unit are entry points, whereas nested procedures and functions are not.

Any entry point defined in the compilation unit whose name matches *identifier* is translated to *string* when inserted into the object file, and hence can only be referenced as *string* when linking. *String* may not contain the NULL character ('*#00').

Any entry points in #USEd libraries and other compilation units whose names match *string* can be referred to within the compilation unit as *identifier*. This also applies to *identifiers* defined by **EXTERNAL** pragmas. **TRANSLATE** pragmas must precede any reference to their identifier.

For example:

```
#PRAGMA TRANSLATE c.routine "c_routine"
#PRAGMA EXTERNAL "PROC c.routine () = 100"
```

#PRAGMA LINKAGE ["*section-name*"**]** *comment*

This directive enables the user to identify modules that he wishes to be placed in on-chip RAM. The user may then prioritise the order in which these modules are linked together by using a linker directive. On-chip RAM is allocated to workspace first and then to code. Provided there is enough RAM available it should be possible for commonly used subroutines to be processed in the on-chip RAM. This should make the program run faster.

Normally the compiler creates the object code in a section named "**text%base**". The **#PRAGMA LINKAGE** directive causes the compiler to change the name of the section to that supplied in the string. If the directive is used but no section name is provided by the user, the compiler supplies the default section name "**pri%text%base**". More than one module may take the section name "**pri%text%base**".

A linker directive is used to change the order in which code modules are linked together, by supplying a list of prioritised *section-name*s, see chapter 19. Provided that the linker does not encounter any linker directives listing *section-names*, it will place "**pri%text%base**" modules first. Any unnamed modules are added in an undefined order at the end of the linked unit.

Note: floating point routines such as **REAL32OP** and **REAL32OPERR** are automatically optimised by the compiler by placing them in a "**pri%text%base**" section.

The **#PRAGMA LINKAGE** directive should appear at the start of the source code, immediately following the **#OPTION** directive, if one is present.

For example:

```
#OPTION "N"
#PRAGMA LINKAGE "PRIORITY1" -- highest priority
```

25.11 INLINE keyword

The keyword **INLINE** may be used immediately before the **PROC** or **FUNCTION** keyword of any procedure or function declaration. This will cause the body of the procedure or function to be expanded inline in any call, and the declaration will not be compiled as a normal routine. Use of **INLINE** procedures or functions may increase the size of the object module but will also avoid the overheads incurred in executing extra calls.

Examples:

```
INT INLINE FUNCTION sum3 (VAL INT x, y, z,) IS  x
                               + (y + z) :


INLINE PROC seterror ()
  error := TRUE
:
```

A call to the FUNCTION sum3:

```
so.write.int(fs, ts, sum3(x,y,z),0)
```

would be expanded by the compiler thus:

```
so.write.int(fs, ts, x + (y + z),0)
```

Note: the declaration is marked with the keyword, but the call is affected. This means that you cannot inline expand procedures and functions which have been declared by a #USE directive; to achieve that effect you may put the source of the routine, marked with the **INLINE** keyword, in a separate file, and include this file with an #INCLUDE directive.

25.12 Implementation of channels

The implementation of channels by the compiler has changed with this issue of the toolset. The data type of a channel is now 'pointer to channel' rather than 'channel'. This means that scalar channels remain the same but arrays of channels become arrays of pointers to channels.

As a result of this **PLACE**ing arrays of channels has also changed.

```
PLACE array.of.channels AT n:
```

now places the array of pointers at that address.

PLACE scalar.channel AT n:

places the channel word at that address.

An example of the placement of channels on links is given in chapter 10.

Arrays of channels may now be constructed out of a list of other channels. For example:

```
PROC p (CHAN OF protocol a, [2]CHAN OF protocol b)
  [3]CHAN OF protocol c IS [a, b[0], b[1]] :
  -- channel constructor

  ALT i = 0 FOR SIZE c
    c[i] ? data
      ...
:
```

Channel constructors extend the existing facilities for manipulating channels, further information is given in chapter 10.

25.13 Implementation of usage checking

This section describes the usage checking that is implemented by the compiler.

25.13.1 Usage rules of occam 2

The usage checking rules of occam 2 are as follows:

- No variable assigned to, or input to, in any component of a parallel may be used in any other component.

- No channel may be used for input in more than one component process of a parallel.

- No channel may be used for output in more than one component of a parallel.

25.13.2 Checking of non-array elements

Variables and channels which are not elements of arrays are checked according to the rules of occam 2.

25.13.3 Checking of arrays of variables and channels

Where possible, the compiler treats each element of an array as an independent variable. This makes it possible to assign to the first and second elements of an array in parallel.

For usage checking to operate in this way, it must be possible for the compiler to evaluate all possible subscript values of an array. The compiler is capable of evaluating expressions consisting entirely of constant values and operators (but not function calls). Where a replicator is used in an expression the compiler can evaluate the expression for all values of the index provided that the replicator's base and count can be evaluated. **Note**: however, that as each iteration of the routine is checked, this can slow the compiler down.

Where an array subscript contains variables, a function call, or the index of a replicator where the base or the count cannot be evaluated, the compiler assumes that all possible subscripts of the array may be used. This may cause a spurious error. For example, consider the following program fragment:

```
x := 1
PAR
  a[0] := 1
  a[x] := 2
```

The compiler reports the assignment to `a[x]` as a usage error. The fragment could be changed to:

```
VAL x IS 1:
PAR
  a[0] := 1
  a[x] := 2
```

This would be accepted by the compiler because `x` can be evaluated at compile time.

The compiler checks segments of arrays similarly to simple subscripts. Where
the base and count of a segment can be evaluated, each segment is treated as
though it has been used individually. Where the base or count cannot be evalu-
ated, the compiler behaves as if the whole array has been used. For example,
the following code is accepted without generating an error:

```
PAR
  [a FROM 4 FOR 4] := x
  a[8] := 2
  [a FROM 9 FOR 3] := y
```

25.13.4 Arrays as procedure parameters

Any variable array which is the parameter of a procedure is treated as a single
entity. That is, if any element of the array is referenced, the compiler treats the
whole array as being referenced. Similarly, if any variable array, or element of a
variable array is used free in a procedure then the compiler treats it as if every
element were used. For example, the compiler reports an error in the following
code because it considers every element of **a** to have been used when **p(a)**
occurred.

```
PROC p([]INT a)
  a[1] := 2
:
PAR
  p(a)
  a[0] := 2
```

Similarly, where one element of an array of channels is used for input or output
within a procedure, the compiler treats the array as if all elements were used in
the same way. For example, the compiler reports an error in the following code
because it considers an output has been performed on every element of **c** when
p() occurred.

```
PROC p()
  c[1] ! 2    -- c free in p
:
PAR
  p()
  c[0] ! 1
```

25.13.5 Abbreviating variables and channels

The compiler treats an element which is abbreviated in an element abbreviation as if it had been assigned to, whether or not it is actually updated. If this causes an apparently correct program to be rejected the program should be altered to use a **VAL** abbreviation. For example, the compiler reports an error in the following code because it considers the first component of the **PAR** to have been assigned to **b**.

```
PAR
  a IS b :
  x := a
  y := b
```

This could be changed to:

```
PAR
  VAL a IS b :
  x := a
  y := b
```

Where a channel is an abbreviation of a channel array element, the compiler behaves as if the whole of the channel array had been used unless the element is an array element with constant subscripts, a constant segment of an array (i.e. with constant base and count) or a constant segment with constant subscripts.

25.14 Implementation of alias checking

This section describes the alias checking that is implemented by the compiler.

25.14.1 Alias checking

In the following Rules 'assigned to' means 'assigned to by assignment or input'.

Scalar variables

(**Rule 1**) If a scalar variable appears in the abbreviated expression of a VAL abbreviation, for example:

```
x   in VAL a IS x + 2 :
```

then that variable may not be assigned to or abbreviated by a non-VAL abbreviation anywhere within the scope of the VAL abbreviation.

(Rule 2) If a scalar variable is abbreviated in a non-VAL abbreviation, for example:

        x   in a IS x :

then that variable may not be referenced anywhere within the scope of the abbreviation.

Arrays

The rules for arrays attempt to treat each element of the array as an individual scalar variable. They allow the maximum freedom possible without introducing run time checking code except at points of abbreviation.

In the following text the word constant means any expression that can be evaluated at compile time.

If an array is referenced in the expression of a VAL abbreviation, for example:

        x   in VAL a IS x[i] :

then the following rules apply to the use of the array within the scope of the abbreviation:

(Rule 3) If the subscript is constant then elements of the array may be assigned to as long as they are only subscripted by constant values different from the abbreviated subscript. Any element of the array may also appear anywhere in the expression of a VAL abbreviation. Any other elements of the array may be non-VAL abbreviated, and run time checking code is generated if subscripts used in the abbreviation are not constant.

(Rule 4) If the subscript is not constant then no element of the array may be assigned to unless it is first non-VAL abbreviated. The non-VAL abbreviation will have to generate run time code to check that it does not overlap the VAL abbreviation. The array may be used in the expression of a VAL abbreviation.

Elements of the array may be accessed anywhere within the scope of the abbreviation except where restricted by further abbreviations.

If an array is abbreviated in a non-VAL abbreviation, for example:

        x   in a IS x[i] :

then the following rules apply to the use of the array within the scope of the abbreviation:

(Rule 5) If the subscript is constant then elements of the array may be read and assigned to as long as they are accessed by constant subscripts different from the abbreviated subscript. Other elements of the array may be abbreviated in further VAL and non-VAL abbreviations, and run time checking code is generated if subscripts used in the abbreviation are not constants.

(Rule 6) If the subscript is not constant then the array may not be referenced at all except in abbreviations where run time checking code is needed to check that the abbreviations do not overlap.

(Rule 7) Variables used in subscripts of the array being abbreviated act as if they have been VAL abbreviated. In the above example 'i' acts as if it has been VAL abbreviated and cannot be altered in the scope of the abbreviation. Where elements of the array being abbreviated are used in the subscript of the array then the abbreviation is checked as if the subscript expression was VAL abbreviated just before the non-VAL abbreviation. For example:

```
a IS x[x[2]] :
```

is checked as if it was written:

```
VAL subscript IS x[2] :
a IS x[subscript] :
```

which (by Rule (6) above) will generate run time checking code.

25.15 Error messages

All messages produced by the compiler are in the standard toolset format.

The compiler libraries are automatically loaded if required, unless the compiler 'E' option is used.

The compiler finds the compiler libraries by searching the path specified by the host environment variable ISEARCH. The most common cause of a compiler library error is failure to set up this logical name correctly.

No object files are generated if an error occurs.

The error messages listed here are those which are produced by incorrect use of the compiler, caused for instance by failing to specify command line options correctly. The compiler also reports all syntax and semantic errors found in the program; these messages are not listed here as they are language specific and therefore outside the scope of this document.

25.15.1 Warning messages

Badly formed #PRAGMA *name* directive

The pragma directive does not conform to the required syntax.

name is not used

The named variable is never used. This warning may be disabled by the NWU command line option.

Name *name* descopes a previous declaration

This name descopes another name which has already been declared. This warning is only enabled when the WD command line option is used.

No compatible entrypoints found in *name*

The named library contains no routines which may be called from this error mode and/or processor type.

Parameter *name* is not used

The named parameter is never used. This warning may be disabled by the NWP command line option.

Placement expression for *name* clashes with interactive debugger

The named variable is placed on one of the transputer links. This may interfere with the INMOS interactive debugging system.

Placement expression for *name* wraps around memory

The calculation of the machine address for this variable has overflowed; the truncated address is used.

Routine *name* imported by multiple #USEs

The named routine exists in two different libraries; an implementation restriction means that this is not permitted.

Routine *name* is not used

The named routine was never called. This warning may be disabled by the NWU command line option.

Run-time disjointness check inserted

number **Run-time disjointness check inserted**

> The compiler has inserted run-time checks to ensure that variables are not aliased (i.e. that they do not overlap). This warning is only enabled when the WO command line option is used.

TRANSLATE ignored: Module containing *name* **has already been loaded**

> The #TRANSLATE pragma must precede any #USE of a library containing that string.

TRANSLATE ignored: Name *name* **has already been used**

> You may not specify multiple translation strings for the same name.

TRANSLATE ignored: String contains NULL character

> The specified string for a #TRANSLATE pragma may not include a NULL (zero) byte.

TRANSLATE ignored: String *name* **has already been used**

> You may not specify multiple names to be translated to the same string.

Unknown #PRAGMA name: *name*

> The pragma name is ignored.

Workspace clashes with variable PLACED AT WORKSPACE *number*

> A variable has been PLACED AT WORKSPACE *number*, and this clashes either with another placed variable, or with the compiler's workspace allocation requirements.

25.15.2 Errors

Bad object file format

> Library or separately compiled procedure object code is not in the correct
> format. The code may not have been linked correctly, or the file may have
> become corrupted.

Badly formed compiler directive

> A compiler directive following # was not recognised.

Badly formed #EXTERNAL directive

> The number of workspace slots to reserve for the call has not been spec-
> ified or negative workspace or vector space slots have been specified in
> error.

Cannot open file "*string*"

> File is missing, or file system error.

Cannot open output file

> The object file could not be opened. File system error.

Cannot open output file (*string*)

> The file given as parameter to the command line R option could not be
> opened.

Cannot open source file

> The source file cannot be opened. Either it does not exist, or there is a
> file system error.

Descriptor has incorrect format

> Library or separately compiled procedure object code is not in the correct
> format. The code may not have been linked correctly, or the file may have
> become corrupted.

Duplicate error modes on command line

> Multiple error modes may not be specified for the compilation.

Duplicate processor types on command line

Multiple processor types may not be specified for the compilation.

Expected string after #COMMENT

`#COMMENT` directive must be followed by a string containing the comment.

Expected string after #OPTION

`#OPTION` directive must be followed by a string containing the options .

'*Filename*' is not a valid object file

Library or separately compiled unit object code is not in the correct format. The code may not have been linked correctly, or the file may have become corrupted.

Files nested too deeply

The maximum depth of nesting permitted within a single compilation unit is twenty.

Code insertion is not enabled

You must use the G or W options to enable assembler inserts.

Instruction is not available in current code insertion mode

You must set the W option in order to use this instruction.

Instruction is not available on target processor

The given instruction is not present in the target instruction set.

Invalid command line option (*string*)

The user specified an unrecognised command line option.

Missing filename

Filename is missing on `#USE`, `#INCLUDE` or `#IMPORT` directive.

Missing object file name

There is no object file name parameter to the command line O option.

Missing output file name

There is no output file name parameter to the command line R option.

No filename given

No source file was specified on the command line.

number **reading source file**

File system error. The source file or an include file could not be read. *number* is the host file system error number.

number **writing to object file**

File system error. The object file could not be written to. *number* is the host file system error number.

Option in illegal position

Only one #OPTION directive is allowed in a file, and it must be on the first non-blank or non-comment line in a file.

Run out of symbol space

The source compilation unit is too large to be compiled.

Unrecognised option "*char***" in option string**

Incorrect compiler options specified after a #OPTION directive.

26 occonf — configurer

This chapter describes the configurer tool occonf 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.

26.1 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.

Linked modules and libraries which are referred to by a configuration description must be already compiled and linked before any file which references them can itself be configured.

The operation of the configurer tool in terms of toolset default file extensions is illustrated below.
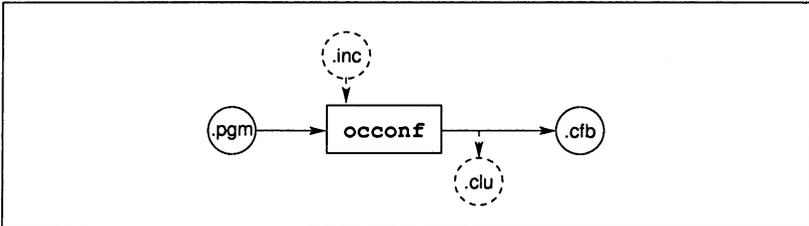
occonf produces two output files; a configuration data file and a subsidiary file which has the extension .clu, this is used by the collector.

26.2 Running the configurer

The configurer takes as input a configuration description file and produces a
configuration data file for input to the collector tool.

To run the configurer use the following command line:

▶   **occonf** *filename* *{options}*

where: *filename* is the configuration description file. If no file extension is speci-
fied, the extension **pgm** is assumed. Only one file may be specified.

options is a list of one or more options from tables 26.1 and 26.2.

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 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:

oc simple
ilink simple.tco hostio.lib -f occama.lnk
occonf simple.pgm
icollect simple.cfb
iserver -sb simple.btl -se

MS-DOS and VMS based toolsets:

oc simple
ilink simple.tco hostio.lib /f occama.lnk
occonf simple.pgm
icollect simple.cfb
iserver /sb simple.btl /se

Option	Description
B	Displays messages in brief (single line) format.
C	Disables the generation of object code. The configurer performs syntax, semantic, alias and usage checking only.
I	Displays extra information as the tool runs. This information includes target and error mode, and information about directives as they are processed. The default is not to display this information.
L	Loads the tool onto the transputer board and terminates.
O *outputfile*	Specifies an output filename. If no output file is specified the configurer uses the input filename and adds the file extension `.cfb`.
R *filename*	Redirects error and information messages to a file.
V	Prevents the configurer from producing code which has a separate vector space requirement. The default is to produce code which uses separate vector space.
Y	Disables interactive debugging with `idebug`.
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.
H	Produces code in HALT error mode. This is the default configuration mode and may be omitted for HALT error mode programs.
S	Produces code in STOP error mode.
X	Produces code in UNIVERSAL error mode.
K	Disables run-time range checking. The default is to insert run-time range checking.
U	Disables the insertion of all extra run-time error checking. The default is to insert run-time error checks. This is a 'stronger' option than **K**, and can be used to implement the occam UNDEFINED error mode.
NA	Disables the insertion of run-time checks for calls to `ASSERT`.
G	Enables the configurer to recognise the restricted range of transputer instructions, via the `ASM` or `GUY` constructs.
W	Enables the configurer to recognise the full range of transputer instructions, via the `ASM` or `GUY` constructs.

Table 26.1 Configurer options

Option	Description
RE	Enables memory lay-out re-ordering.
NWP	Do not warn if parameters are not used.
NWU	Do not warn if variables or routines are not used.
WD	Provides a warning whenever a name is descoped.
WO	Provides a warning whenever a run-time alias check is generated.
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 26.2 Configurer options

26.2.1 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 2.10.3.

26.3 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 12.7.

26.4   Configuration error modes

The configuration error mode determines the behaviour of a program if it fails during execution. The execution behaviour of programs configured in the different modes is as follows:

HALT   An error halts the transputer immediately.

STOP   An error stops the process and causes graceful degradation.

UNIVERSAL   Code configured in this mode behave as either HALT or STOP mode, according to the state of the transputer's *HaltOnError* flag.

The error mode selected for the configuration must be compatible with the error mode of the compiled units, referenced by the configuration source. The configurer will produce an error message, if this is not the case.

Table 26.3 indicates the compilation error modes which are compatible and the possible error mode they may be configured for.

Compatible compilation error modes	occonf options
HALT, UNIVERSAL	H
STOP, UNIVERSAL	S
UNIVERSAL	X

Table 26.3 `occonf` error modes

Compilation error modes and their effect are described in more detail in section 4.4.

Note: that occam UNDEFINED mode can be achieved by using the configurer 'U' option, to disable the insertion of run-time checks. This option behaves in the same way as the 'U' option to the occam compiler, which is documented in section 4.4.

26.5   Enable/Disable Error Detection

By default the configurer inserts code to execute run-time checks for errors it cannot detect during configuration. In some circumstances it may be desirable to omit the run-time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes. Three command line options are provided to enable the user to control the degree of run-time error detection performed; they are the 'K', 'U' and 'NA' options.

The 'K' option disables the insertion of run-time range checks on array subscripting and array lengths.

The 'U' option prevents the configurer from inserting any code to explicitly perform run-time checks. This option will disable run-time checks associated with type conversion, shift operations, array access, range validation and replicated constructs such as SEQ, PAR, IF, and ALT.

The 'NA' option prevents the configurer from inserting any code to check calls to ASSERT. In effect, each ASSERT behaves like SKIP. Any calls to ASSERT which can be evaluated during configuration will still be checked.

Note: that some checks are still performed; some transputer instructions implicitly check for erroneous conditions.

The K, U and NA options behave in exactly the same way, as the same options provided for the occam compiler. The effect of using these options is described in detail in section 4.4.1.

26.6   Enabling memory lay-out re-ordering

The 'RE' option enables the user to have more control of the layout of code and data areas in memory. When this option is used, the special processor attributes 'order.code' and 'order.vs' can be used to indicate the relative priority of different data areas, see section 5.5.3. **Note**: use of this option means that the INMOS debugger cannot be used, neither in interactive mode nor in postmortem mode.

26.7 Enabling/disabling warning messages

There are four command line options which allow the user to either enable or disable the generation of certain warning messages by the configurer:

- The 'NWP' option disables warning messages being generated when parameters to procedures are not used.

- The NWU option disables warning messages being generated when variables or routines are not used.

- The WD option provides a warning whenever a name is descoped, for example when a name is used twice and one occurrence of it is hidden within an inner procedure. See section 8 of the '*occam 2 Reference Manual*' for details of occam scope rules.

- The WO option provides a warning whenever a run-time alias check is generated i.e. to check that variables do not overlap. These checks generate extra code and the user may wish to be alerted to this.

Section 26.10 lists the various warning messages which are affected by these options.

26.8 Support for interactive debugging

Interactive debugging is supported by default. When interactive debugging is enabled the configurer will generate calls to library routines to perform channel input and output, rather than using the transputer's instructions. Interactive debugging must be enabled in order to use the interactive features of the debugger.

Interactive debugging may be disabled by using the configurer 'Y' option. This option forces the configurer to use sequences of transputer instructions for channel input and output, resulting in faster code execution.

Code which has interactive debugging disabled may call code which has interactive debugging enabled, but *not* vice versa. However, when interactive debugging is disabled in one part of the program this will prevent the interactive features of the debugger being used on the program as a whole.

26.9   ASM and GUY code

Two configurer options are provided to enable the configurer to recognise trans-
puter instructions, via the ASM and GUY constructs. The 'G' option enables a
limited range of instructions, which are listed in part 2, appendix B. The 'W' op-
tion enables the full range of instructions. For further details of the low level
programming supported by the toolset see chapter 10.

26.10 Configurer diagnostics

If the source code does not conform to the occam 2 language definition, then
the configurer will issue diagnostics, in the form of error messages, during the
compilation process. When this occurs no object file nor configuration binary file
will be produced.

Errors in the configuration source produce diagnostic messages in standard
toolset format. Details of the format can be found in section 2.12.1.

Diagnostics are generated at severities *Warning*, *Error*, and *Fatal*.

Diagnostic messages are listed in the following sections by severity.

26.10.1 Warning messages

The following diagnostic messages are generated at severity level *Warning*.

Badly formed #PRAGMA *name* **directive**

The pragma directive does not conform to the required syntax.

name **is not used**

The named variable is never used. This warning may be disabled by means of the NWU command line option.

Name *name* **descopes a previous declaration.**

This name descopes another name which has already been declared. This warning is only enabled by means of the WD command line option.

Parameter *name* **is not used**

The named parameter is never used. This warning may be disabled by means of the NWP command line option.

Placement expression for *name* **clashes with interactive debugger**

The named variable is placed on one of the transputer links. This may interfere with the INMOS interactive debugging system.

Placement expression for *name* **wraps around memory**

The calculation of the machine address for this variable has overflowed; the truncated address is used.

Routine *name* **is not used**

The named routine is never called. This warning may be disabled by means of the NWU command line option.

Run-time disjointness check inserted or

*number*Run-time disjointness checks inserted

The configurer has inserted run-time checks to ensure that variables are not aliased (i.e. that they do not overlap). This warning is only enabled by means of the **WO** command line option.

TRANSLATE ignored: Name *name* has already been used

Multiple translation strings may not be specified for the same name.

TRANSLATE ignored: String contains NULL character

The specified string for a TRANSLATE pragma may not contain a NULL (zero) byte.

TRANSLATE ignored: String *name* has already been used

Multiple names may not be specified to be translated to the same string.

TRANSLATE ignored: Module containing *name* has already been loaded

The TRANSLATE pragma must precede any #USE of a library containing that string.

Unknown #PRAGMA name: *name*

The pragma name is ignored.

Workspace clashes with variable PLACED AT WORKSPACE *number*

A variable has been **PLACED AT WORKSPACE** *number*, and this clashes either with another placed variable, or with the configurer's workspace allocation requirements.

Index

inmos