occam 2 toolset user manual

INMOS Limited

72 TDS 184 00

INMOS document number: 72 TDS 184 00

Contents overview

User Guide

Reference Manual

Appendices

Contents

Preface

This manual is a combined user and reference guide to the occam 2 toolset.

The occam 2 toolset is a set of software tools for developing transputer programs 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 syntax checker, a librarian tool for building code libraries, a network debugger for analysing halted programs, 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 transputer parallel processing. Programs can also be written in C, FORTRAN, and Pascal 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 special software. For details of the products available to support EPROM programming, contact INMOS.

1 How to use the manual

1.1 About the manual

This manual is in two parts: a *User Guide* showing how the tools are used to develop programs on single transputers and transputer networks; and a *Reference Manual* containing details about the individual tools. Reference material for programmers such as predefined names and constants, transputer instructions, and technical information about the software can be found in the appendices at the back of the book. A glossary of terms and a short bibliography of referenced material is included.

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.

Examples are used throughout the book to illustrate how the tools are used during program development. All examples are coded in occam 2.

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.

1.1.1 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 transputer evaluation board on which the programs will be developed. Information about INMOS transputer evaluation boards is available in the form of product datasheets.

1.2 User guide

The User Guide 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.

The Guide begins with an introduction to transputers and occam programming and an overview of the toolset. It goes on to describe how to build programs for single and multiple transputers, shows how to debug them using the debugger and simulator tools, and outlines how to access host services such as the file system. Further chapters in the User Guide describe how to use C, FORTRAN, and Pascal in transputer programs and explain some low level programming features of occam such as the placement of variables and the insertion of instruction code.

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

1.2.1 Getting started

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 3, which contains an overview of the toolset.

Before attempting to write any programs of your own you should read chapter 4 and chapter 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 T414 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 programs written in C, FORTRAN, and Pascal can be inserted into transputer programs using an occam wrapping and special interface code. Read and digest the information in this chapter carefully before attempting to write mixed language programs.

1.3 Reference manual

The Reference Manual contains reference information for all tools in the toolset, plus details of the occam libraries. Each tool is described in a separate chapter.

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

1.4 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.
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 a UNIX based toolset and the line containing the '/' character if you have a non-UNIX based toolset.

2 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.

2.1 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 INMOS C, FORTRAN and Pascal compilers the occam 2 toolset can be used to build and debug mixed language software systems.

Multi-processing is now widely accepted as the only way to substantially increase system performance. Transputers and the occam 2 toolset make building high performance parallel systems as simple as sequential programming with conventional microprocessors.

2.2 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware. They can be connected together in any configuration, and form a building block 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 2.1 shows the architecture of the transputer.

Multi-transputer systems can be built very simply. Each transputer has four high speed communication links operating at 10 to 20 Mbits per second that allow transputers to be connected to each other in any configuration. The circuitry to drive these links is all on the transputer chip, and it takes just two wires to connect two transputers together. Figure 2.2 shows four transputers connected using their communication links, and the communication paths between them.

In addition to providing a communication link between processors, transputer links allow memory to be examined without loading a program, and permit pro-

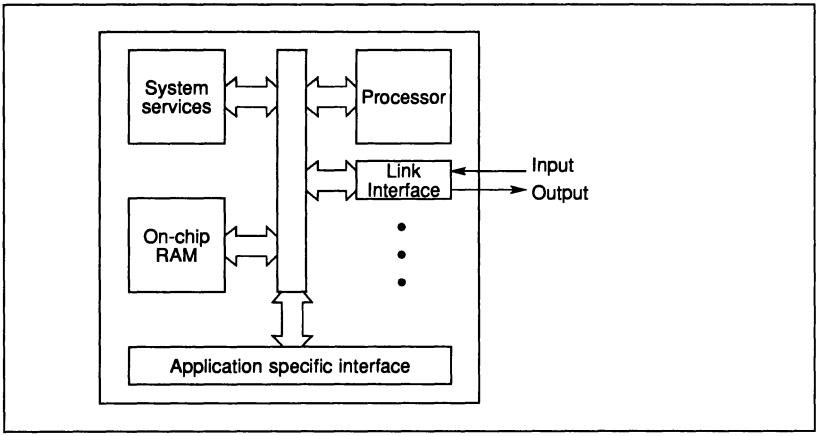


Figure 2.1 Transputer architecture

grams to be loaded and executed. This allows whole networks of transputers to be loaded down a single transputer link.

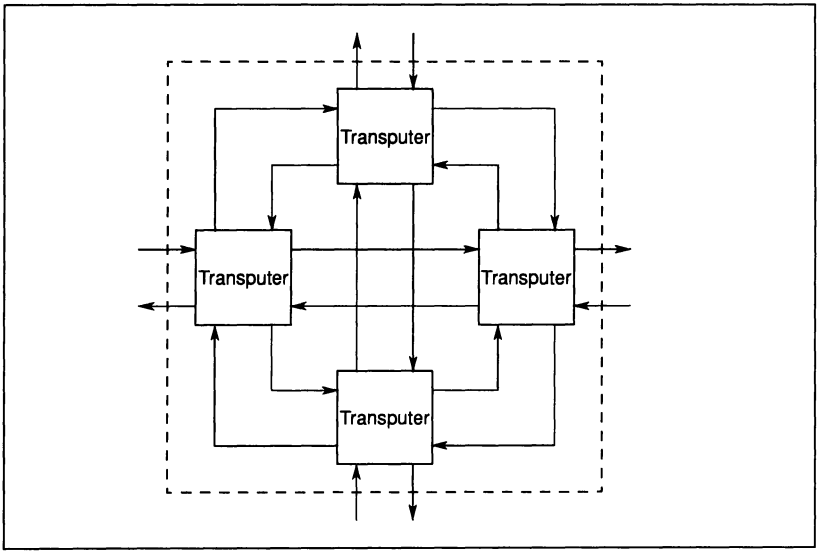


Figure 2.2 A node of four transputers

Each single transputer supports parallel processing through a system of internal channels implemented as words in memory. Each transputer has a highly effi-

cient 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.

There is a complete family of transputer devices, including: 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.

2.3 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, FORTRAN, and Pascal 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.

2.3.1 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 C, FORTRAN and Pascal 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.

2.3.2 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 2.3 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.

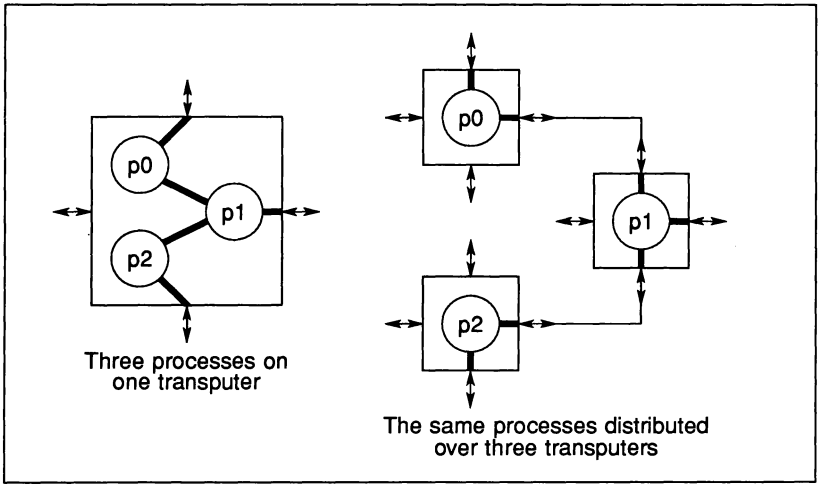


Figure 2.3 Mapping processes onto one or several transputers

2.3.3 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.

For details of occam 2, see the *'occam 2 Reference Manual'*, which contains definitive information about the language. For those with no knowledge of occam but who are already familiar with a high level language, *'A tutorial introduction to occam programming'* provides a good introduction to occam.

2.3.4 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
- Placement of variables at specific addresses in memory, so that memory mapped devices can be accessed.

Detailed discussions of the some of technical issues involved in transputer programming can be found in the INMOS Technical Notes listed in the bibliography to the rear of this manual.

2.4 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 this reason, no editor or Make program is provided with the toolset. For embedded systems, programs can be loaded onto the target hardware from the host via a transputer link.

2.4.1 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 compiler before transferring it to hardware.

2.4.2 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. Validated source can be created easily with the toolset syntax checker, and binary code files can be collected together either for code sharing or for convenience when creating code libraries.

Code for single transputers 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 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.

2.4.3 Debugging

occam programs can be executed and tested without transputer hardware using the T414 simulator tool which provides full symbolic debugging functions. This method is appropriate for debugging individual parts of a large transputer program.

Programs for multi-processor systems can be configured to run on a transputer evaluation board, and then loaded and debugged using the symbolic network debugger that allows a halted program to be analysed in terms of its source code.

User guide

72 TDS 184 00

April 1989

3 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.

3.1 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 3.1.

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 workstation running SunOS.

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

3.2 Program development

Stages in the program development process are outlined below and the tools to use at each stage are listed.

1. **Coding:** Program modules are written, and then checked for correct syntax before compilation using the occam 2 syntax checker **icheck**.

2. **Compilation:** Individual components and modules are compiled using the occam 2 compiler **occam**. Libraries of compiled code can be created using the librarian tool **ilibr**.

3. **Linking:** Components of a program, such as processes running on individual processors, are linked together with libraries and other separately compiled units using the linker tool **ilink**.

Program	Description
iboot	The bootstrap tool. Produces bootable code for single transputers.
icheck	The occam 2 syntax checker.
iconf	The configurer. Builds bootable code for multitransputer programs.
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.
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 T414 transputer simulator.
iskip	The skip loader tool. Prepares transputer networks to run programs without using the root transputer.
occam	The occam compiler. Compiles source for IMS T212, M212, T222, T414, T425 and T800 transputers.

Table 3.1 The occam 2 toolset

4. **Creating executable code:** For single processor programs code that can be directly loaded onto the transputer and run (**bootable** code) is produced by adding bootstrap information to the linked code, using the bootstrap tool **iboot**. For multitransputer programs bootable code is generated from a configuration description file using the configurer tool **iconf**. This produces bootable code for a specific arrangement of transputers.

5. **Loading and running:** Programs are loaded onto transputers or transputer networks using the hos file server tool **iserver**. The **iskip** tool can be used to load programs onto external networks over the root transputer.

The server tool provides run time support for interfacing with the host.

6. **Debugging:** If a program does not work correctly it may be debugged using the debugger **idebug** or the T414 simulator **isim**. Both tools leave the object code unmodified. The binary listing tool **ilist** can be used to display information about object code.

7. **Recompilation:** The code is modified after debugging, recompiled and retested. The Makefile generator **imakef** can assist in regenerating code during program development.

3.3 The toolset

This section provides a brief introduction to each of the tools in turn.

3.3.1 iboot – the bootstrap tool

The bootstrap tool adds bootstrap code to linked programs for single processors. Bootstrap code is required for programs in order to initialise the processor (when booting via a link), and to load the program.

3.3.2 icheck – the occam 2 syntax checker

The checker performs a comprehensive check of occam 2 syntax including declared but unused variables. Using the regular structure of occam the checker is able to recover from syntax errors and detect multiple errors in the occam source. It produces more informative messages than the compiler and can be used to filter out syntax errors prior to compilation.

3.3.3 iconf – the configurer

The configurer generates executable code for multitransputer networks from occam program units. Using an occam configuration description file that describes the placement of code modules on a transputer network, it produces a file containing bootable code and network loading data. The file can then be directly loaded and run on a transputer network.

3.3.4 idebug – the debugger

The debugger provides post mortem debugging of occam programs. It permits complete analysis of occam code and transputer hardware following a run-time error, using symbolic and low level facilities. The debugger provides limited

support for C, FORTRAN and Pascal programs.

3.3.5 `idump` – the memory dumper

The **idump** tool writes the contents of the root transputer's memory to a disk file that can be read by the debugger. This tool is used when debugging programs that use the root transputer, because the debugger overwrites the root transputer's memory.

3.3.6 `ilibr` – the librarian

The librarian collects compiled code files into libraries. Each separately compiled file becomes a library module that can be selectively linked.

3.3.7 `ilink` – the linker

The linker links together separately compiled units into a single file, resolving all external references.

3.3.8 `ilist` – the binary lister

The binary lister tool displays in a readable form the contents of object code files. Options control the type of information to be displayed.

3.3.9 `imakef` – the Makefile generator

The Makefile generator creates Makefiles for input to MAKE programs. Given the name of a program or library to be built, it traces references to other code and library files and compiles a list of file dependencies and compilation commands for input to the **Make** utility. The C source of the **imakef** program is supplied and can be modified to suit any MAKE program.

The Makefile generator is also used to create library usage files.

3.3.10 `iserver` – the host file server

The server tool has two functions. Firstly, it loads bootable programs onto transputers or transputer networks. Secondly, it provides the run-time environment that enables application programs to communicate with the host.

3.3.11 isim – the T414 simulator

The T414 simulator simulates the operation of the T414 transputer, enabling programs to be tested and debugged in the absence of hardware. It provides debugging features such as the inspection of variables, registers, and queues, disassembly of memory, break points, and single step execution.

3.3.12 iskip – the skip loader tool

The skip tool sets up the environment that allows programs to be loaded and run on networks that do not incorporate the root transputer. The tool is used to assist with the loading of programs onto external networks through a root transputer and to allow the debugger to run on the root transputer without overwriting program code.

3.3.13 occam – the occam 2 compiler

The occam 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, T222, M212, T414, T425 and T800 transputers in four 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
- #IMPORT - imports code modules written in C, FORTRAN, and Pascal. The module must be compiled using an INMOS or INMOS-compatible compiler.

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

3.4 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.

Some libraries are supplied as object code, others as both object and source code. Table 3.4 lists the libraries that are supplied with the toolset and specifies how they are supplied. Details of all the libraries can be found in chapter 24.

Library	Description	Format
`occam`*xx*`.lib`	Compiler libraries	object
`hostio.lib`	general purpose i/o library	both
`streamio.lib`	Stream i/o support	both
`snglmath.lib`	single length maths functions	both
`dblmath.lib`	double length maths functions	both
`tbmaths.lib`	T414/T425 optimised maths functions	both
`string.lib`	String handling routines	both
`xlink.lib`	Extraordinary link handling routines	object
`convert.lib`	Type conversion routines	both
`crc.lib`	CRC coding	object
`process.lib`	Board and process support	both

3.4.1 Constants

Files containing definitions of constants and protocols are also provided for use with the occam libraries. These are listed in table 3.2.

File	Description
`hostio.inc`	Host file server constants
`streamio.inc`	Stream i/o constants
`mathvals.inc`	Mathematical constants
`linkaddr.inc`	Transputer link addresses

Table 3.2 Library constants

3.4.2 Compiler libraries

The compiler libraries are used internally by code generated by the compiler. With a number of exceptions which are outlined in section 24.2.1, they are not intended for direct use by the programmer. The compiler references them automatically by searching the directories specified by the `ISEARCH` host environ-

ment variable.

The compiler automatically loads the library required for a specific combination of compiler options. The libraries can be disabled by giving the compiler '**E**' option, and *must* be disabled for transputer classes TA or TC and UNIVERSAL mode. For details of transputer classes see section 4.4.1.

3.4.3 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 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**.

3.4.4 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 communicate with the server via the **SP** protocol either directly or through a buffer or multiplexor routine which itself uses 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 buffering and multiplexing, and protocol conversion.

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**.

The streamio routines perform the same function as those used in the IMS D700D Transputer Development System (TDS), and enable programs to be ported between the toolset and the TDS.

Many of the streamio routines cannot communicate directly with the server and must be passed to another process for forwarding to the server. Procedures for converting screen and keyboard stream protocols to the host file server protocol **SP** are provided for this purpose.

3.4.5 Other libraries

String handling library

The string handling library provides string handling functions and procedures, 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 link errors on a channel.

Block CRC library

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

Process library

The process library provides two routines: a serial port driver for INMOS transputer evaluation boards; and a debug timer to assist with debugging deadlocked programs.

3.5 Implementation differences

The toolset is available for three different host machines and operating systems:

- IBM PC running DOS
- DEC VAX running VMS
- Sun Microsystems Sun-3 running SunOS (UNIX)

Source and binary code generated by the toolset is portable across these three systems.

Host operating system dependencies such as differences in file handling and command line syntax are described below.

3.5.1 Host dependencies

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

Command line syntax

The major difference between different host implementations is the option prefix character. For UNIX based toolsets the prefix character is the hyphen '-'; for the IBM PC and VAX toolsets it is the forward slash character '/'.

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 chapter 24.

Directories and files

A directory path searching mechanism is implemented within the toolset, and full pathnames need not be given. For details of the mechanism, see section 3.7.4.

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

3.6 Host environment variables

The toolset assumes four environment variables on the host system. These are listed below.

Variable	Meaning
`ISEARCH`	The list of directories on which the toolset will search for files if the full pathname is not specified.
`ITERM`	The file containing terminal keyboard and screen codes.
`IBOARDSIZE`	The memory size of the transputer board.
`TRANSPUTER`	The address at which the transputer board is connected to the host if not at the default address.

The mechanism used to define these variables varies from host to host. For example, on the IBM PC running DOS they are defined as environment variables using the `set` command and on VAX systems running VMS they can be set up as logical names or VMS symbols. For details of how to define host environment variables on your system, see the Delivery Manual that accompanies the release.

3.7 Toolset conventions

All tools in the toolset, and all implementations of the toolset, use a common set of conventions and defaults.

The toolset uses conventions in the following areas:

- Command line syntax
- Command line options
- File naming
- File location
- Error handling and error message format.

3.7.1 Command line conventions

Syntax

All tools in the toolset conform to the following command line conventions:

- Case is ignored, except for command names on UNIX systems.
- Options must be prefixed by the option prefix character ('-' for UNIX based toolsets, '/' for VAX-VMS and IBM PC-DOS based toolsets).
- If an option takes more than one parameter the parameters must be enclosed in parentheses `()`, and separated by commas.
- Options can occur in any order after the filename.
- Spaces between options and the case of letters in parameters are not significant.

Common 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 '`I`' option, where supported, displays progress information as the tool runs.
- 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 '`L`' option, where supported, loads the tool onto a transputer board without loading the program. It can be used to test for the existence of a particular tool on the system.
- 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.

3.7.2 Filename conventions

Filename conventions are used for three reasons. Firstly, they enable filenames to be used in a host independent manner. Secondly, they allow file extensions

to be omitted in many commands, because defaults can be assumed. Lastly, they enable the debugger and Makefile generator tools to trace source files for a program.

Filenames

Filenames must not contain the characters: dot `.`, colon `:`, semi-colon `;`, square brackets `[]`, round 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.

File extensions

The toolset uses standard file extensions to specify types of files. Where possible the tools assume these conventions unless otherwise directed. Use of these conventions is recommended, and is required if the Makefile generator **`imakef`** is used to assist with version control. For example, the occam compiler **`occam`** assumes the extension **`.occ`** on the input file, and the configurer tool **`iconf`** assumes a **`.pgm`** extension. Other tools such as **`ilink`** which cannot make assumptions about the input file to use require the extension to be explicitly stated. Filenames and extensions created by the toolset are always generated in lower case.

File extensions used by the occam toolset and the tools to which they relate are given in table 3.3.

Where the final two characters in the file extension are not fixed, for example **`.b`***xx*, they reflect the compiler options that were used when the program was compiled, and indicate the transputer type and error mode of the object code. Characters and their meanings are given in table 3.4.

A complete map of the toolset file extensions and their relationships to the main tools is shown in figure 3.1.

3.7.3 Locating files

The tools locate files by searching a specified *directory path* on the host system. The path is specified using the host environment variable **`ISEARCH`**. For details of path searching mechanisms on specific hosts, see section 3.7.4 and the Delivery Manual that accompanies the shipment.

Extension	Tool	File Type	Description
.btl	iconf	Output	Loadable code for boot from link boards.
.btr	iconf	Output	Loadable code minus bootstrap information. Used for EPROM support.
.b*xx*	iboot	Output	Bootable code for single transputer.
.c*xx*	ilink	Output	Linked code.
.dsc	iconf	Output	Configurer code descriptor file. ASCII format.
.dmp	idump	Output	Memory dump file. Read by the debugger.
.d*xx*	iboot	Output	Bootable code descriptor file. ASCII format.
.inc	occam	Input	Predefined constants and protocols.
.lbb	ilibr	Input	Library indirect file.
.liu	imakef	Input	Library usage file.
.lib	ilibr	Output	Library file. Compiled code.
.l*xx*	ilink	Input	Linker indirect file.
.map	iconf	Output	Configuration map. ASCII format.
.m*xx*	ilink	Output	Module map. Binary format.
.occ	occam	Input	occam source.
.pgm	iconf	Input	Configuration description source file.
.r*xx*	iboot	Output	Single transputer code with no bootstrap.
.s*xx*	ilink	Output	Linker symbol table. ASCII format.
.t*xx*	occam	Output	Compiled code.

N.B. for extensions .b*xx*, .c*xx*, etc., the value of *xx* depends on the transputer type (**2**, **4**, **5**, **8**, **a**, **b**, **c**) and the error mode (**h**, **s**, **u**, **x**).

Table 3.3 Toolset file extensions

The tools conform to the following search rules:

1 If the filename contains a directory specification then the filename is used as given. Relative directory names are treated as relative to the directory in which the tool is invoked.

2 If no directory is specified the directory in which the tool is invoked is

2nd character	
2	T212, T222, M212
4	T414
5	T425
8	T800
a	T800, T425, T414
b	T425, T414
c	T800, T425

3rd character	
h	Halt mode
s	Stop mode
u	Undefined mode
x	Universal mode

Table 3.4 File extension characters

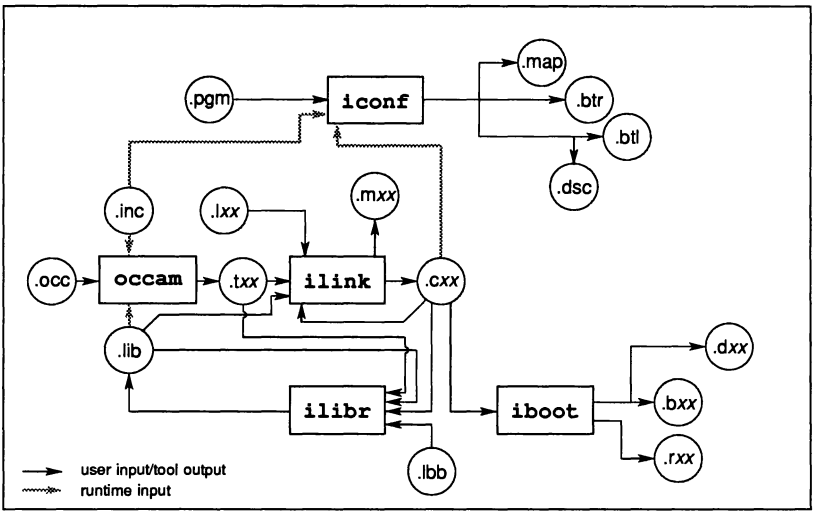


Figure 3.1 Relationships between tools and file extensions

assumed.

3 If the file is not present in the current directory, the path specified by the environment variable (or logical name) **ISEARCH** is searched. If there are several files of the same name on this path, the first occurrence is used.

4 If the file is not found using the above rules, then the file is assumed to be absent, and an error is reported.

If no search path has been set up then only rules 1 and 2 apply.

All files are written to the current directory.

3.7.4 Search paths

Directories to be searched are specified as a list of search paths. On the IBM PC and Sun-3 systems search paths are specified using the **ISEARCH** host environment variable; on VAX systems paths are specified using the logical name **ISEARCH**. Directories are searched in the order that they appear in the list.

Directory paths must be terminated by the appropriate directory separator character for the system you are using ('\' for non-UNIX based toolsets and '/' for UNIX-based toolsets), and entries in the list must be separated by a space or a semi-colon.

For more details about how to set up search paths see the Delivery Manual.

3.7.5 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 tools can generate two types of messages: *Error* and *Warning*.

Errors are faults from which recovery is not possible, and the tool aborts without performing the specified action. Error messages are prefixed with **Error:**.

Warnings identify inconsistencies, or warn of impending errors; the tool continues and does not abort. Warning messages are prefixed with **Warning:**.

Message formats

Errors generated by most tools in the toolset are displayed in the following formats:

Error–*toolname*–*filename* (*linenumber*)–*message*

Warning–*toolname*–*filename* (*linenumber*)–*message*

filename and *linenumber* are optionally displayed where they are meaningful for the error, for example, where the format of a file is incorrect or the file has been corrupted.

Two tools generate messages in special formats. These are:

- The debugger **idebug**
- The simulator **isim**.

The formats will become familiar with use.

4 Programming single transputers

This chapter provides an introduction to occam programming using the toolset, using example programs for single processors. For information on programming multitransputer networks see chapter 5.

Before reading this chapter you 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

Section 4.3 contains a simple programming example to get you started. A more complex example, illustrating separate compilation, can be found in section 4.9.

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 T414 simulator tool **isim**.

4.2 occam programs

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

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

The procedure can have any legal occam name. You must always supply the procedure with the formal parameters 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 chapter 24. 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 vector **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 workspace from the value specified in the **IBOARDSIZE** host environment variable. The **memory** vector is passed to the program as a vector of type INT, where it can be used. By allowing programs to be run on boards with different memory sizes, this vector aids program portability between different boards.

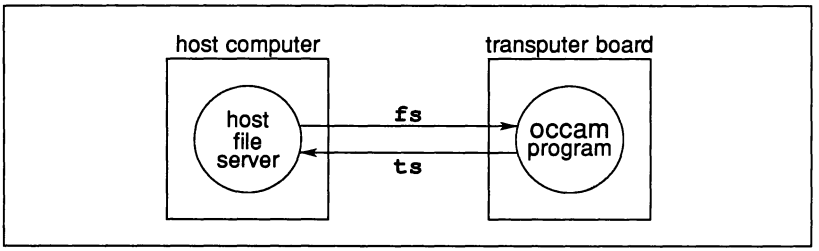


Figure 4.1 Program input/output

4.2.1 Checking programs

The occam syntax checker, **icheck**, can be used to check the syntax of occam programs. When a program is compiled the syntax is checked automatically; however, it is often faster to use the checker to eliminate syntax errors before running the compiler. The checker does a more comprehensive syntax check than the compiler, and reports every error. The checker can be applied to any file containing occam source.

4.2.2 Compiling programs

The compiler is capable of compiling code for any one of a range of transputers (the IMS T212, M212, T222, T414, T425 and T800) in four error modes. The three standard error modes are HALT system, STOP process, and UNDEFINED. A special fourth mode, UNIVERSAL, is provided to allow mixing of code. 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 23.2 for a full description of these options.

If the compiler detects an error, the file name and line number of the error are displayed along with a message explaining the error. If an error is found the compilation is aborted.

If the compilation succeeds, the compiler creates a new code file. The filename for the new file is derived from the name of the source file with the appropriate extension added. The filename can also be specified on the command line.

Compilation information

It is sometimes necessary to check how much code has been generated by a compilation, and how much workspace (data space) will be required to run the code. This information is stored in the code produced by the compiler, linker and librarian. To display the information use the binary lister tool **ilist**. For details see chapter 18.

4.2.3 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.

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. The correct library for your program depends on its transputer type and error mode. Separate libraries are provided for the T212, T414/T425, and T800 in HALT, STOP, and UNDEFINED error modes. For a list of the compiler libraries see section 24.2.

4.2.4 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, entry point data, external references within the code, and debugging data. For more details see chapter 18.

4.2.5 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 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 the bootstrap code added, by the bootstrap tool **iboot**. In addition the bootstrap tool checks that the program parameters are correct, as described in section 4.2.

4.2.6 Loading and running programs

Bootable programs can be loaded onto the transputer evaluation board using the host file server **iserver** (see chapter 20).

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, use the server '**SE**' option.

Programs can also be loaded onto transputer networks, without running code on the root transputer, by first using the **iskip** tool to set up a skip process and then loading the program in the normal way using **iserver**. This can be useful when loading programs onto external networks via a transputer evaluation board, and for debugging programs that use the root transputer to run all or part of a program. For details of skip loading see section 6.5.

4.2.7 Interrupting programs

To interrupt a program while it is still running, press the host system break key to interrupt the server. 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**'.

4.3 Compiling a simple example program

This section contains a tutorial that shows you how to compile, link, and run a simple example program. The tutorial assumes that you have a boot from link board containing a IMS T414, T425, or T800 processor. If you have a board fitted with any other transputer you must compile the program for that transputer type and use the correct file extension in the subsequent command lines.

If you do not have a transputer board use the T414 simulator tool **isim** from this point onwards.

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.

The server library **hostio.lib** is referenced by the **#USE** directive. This library contains all the procedures used by the program. See chapter 24 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.

4.3.1 Checking the example program

The program should be checked before being compiled. To check the program type:

```
icheck simple
```

You need use no options as you are going to run the program on an IMS T414 transputer in **HALT** system mode, which is the default. Although the checker does not produce code it must know the processor type in order to check the library references. Options always follow the filename and must be preceded by the option prefix character ('-' for UNIX, '/' for other toolsets).

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

If any errors are found the checker displays an error message for each error it finds. The error message indicates the nature of the error, the line, and position on the line where it was detected. If no errors are found the checker terminates and returns you to the operating system.

4.3.2 Compiling the example program

Having checked the program syntax and found it correct the program can be compiled. To compile the program type:

```
occam simple
```

If you used any options with the checker you must use the same options again with the compiler. The compiler will create a file called **simple.t4h**, containing the code produced by the compilation.

4.3.3 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.t4h hostio.lib
```

The linked program will be written to the file **simple.c4h**.

Note: In more complex programs libraries may be dependent on other files and libraries. To ensure all necessary libraries are linked into a program, use **imakef** and a suitable MAKE program. For more details about **imakef** see chapter 19.

4.3.4 Running the example program

Before the program can be run it must be made 'bootable'. To do this use the bootstrap tool **iboot**. Type:

```
iboot simple.c4h
```

The bootable program will be written to the file **simple.b4h**. Chapter 11 gives more information on the bootstrap tool.

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

```
iserver /sb simple.b4h

iserver -sb simple.b4h
```

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. For more details about the server options see chapter 20.

If you are using the simulator to run the example program type:

```
isim simple.b4h
```

When the simulator enters the 'Monitor page', use the '**X**' command to generate bootable code and then use the '**G**' command to start the program.

For more details about how to use the simulator see chapter 21.

4.4 Compiler parameters

This section explains the meaning of the compiler parameters and how they are used during compilation.

The toolset occam compiler produces code for any one of a range of transputer processors and in any one of a number of error modes. The transputer type and error mode must always be supplied unless the compilation is for an IMS T414 in HALT mode, as these are the defaults.

There are a number of other compiler functions that may be controlled by compiler options, such as preventing the use of the compiler libraries.

The main compiler options are listed in Table 4.1. For further details see section 23.2. By invoking the compiler with no filename or options you can obtain a brief summary of the options.

Option(s)	Description
T212 T2	T212
M212 T2	M212 (same as T212)
T222 T2	T222 (same as T212)
T414 T4	T414
T425 T5	T425
T800 T8	T800
TA	T414, T425, T800
TB	T414, T425
TC	T425, T800
H	HALT mode
S	STOP mode
U	UNDEFINED mode
X	UNIVERSAL mode
N	No usage checking
A	No alias checking
E	Disable the compiler libraries
V	No separate vector space

Table 4.1 Compiler options

4.4.1 Compilation for different transputers

The compiler produces code targetted at a particular transputer type. All compilations for a single processor must be for the same or a compatible transputer

type.

Transputer classes

The compiler can produce code that will run on different transputers by taking advantage of commonality in their instruction sets. Provided that no code is written which compiles into instructions which are not shared between different processors, the code will run normally.

The commonalities that exist between different processors are as follows:

- T414 and T425 transputers share the same instruction set except for CRC and 2D block move operations.
- T425 and T800 transputers share the same set except for floating point operations.
- T414, T425, and T800 transputers share the same set except for CRC, 2D block move, and floating point operations.

These groupings form the basis of transputer **classes**. The three classes and the target code with which they are compatible are listed below.

Class	Compatible code
TA	T800, T425, T414
TB	T425, T414
TC	T800, T425

All 16-bit transputers (IMS T212, T222, M212) share the same instruction set and there is no overlap with 32-bit transputers.

Code compiled for a transputer class must be able to run on any member of that class. If the source code compiles into transputer code that is not common for all members of the class then an error is reported.

For example, code compiled for class TC cannot contain floating point and extended arithmetic because operations on **REAL** numbers are implemented differently on the two machines. On the T425 the implementation is in software whereas on the T800 it uses the on-chip floating point processor. Similarly, code compiled for class TB can contain no CRC or 2D block move operations because the respective transputer instructions are not implemented on the T414. Code compiled for class TA can contain no floating-point, CRC, or 2D block move operations.

The restrictions on floating point arithmetic apply only to *operations* on the variables, or the returning of a **REAL** result from a function, because these cause

dissimilar instructions to be used. The declaration of **REAL** variables and the passing of **REAL** parameters into procedures or functions is not prohibited.

4.4.2 Mixing code for different transputers

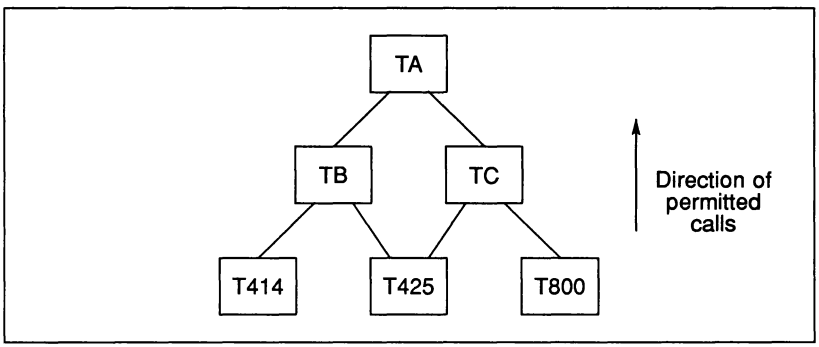
By using transputer classes for compilations it is possible to produce code that may be mixed with code for other transputer types and classes. It should be noted that this not possible for all compilations, but only where instruction sets overlap.

The rule for mixing code is as follows:

> Code may be called provided it is compiled for a class which is the same or is a superset of the calling code.

The code that can be run on different processor types is listed below.

Processor	Compatible code
T2 series	T212
T800	T800, TC, TA
T425	T425, TC, TB, TA
T414	T414, TB, TA
TC	TC, TA
TB	TB, TA
TA	TA



When compiling for a transputer class the compiler will report an error if the source is such that it cannot be compiled for that class. This will often take the form of an undeclared procedure or function from the compiler libraries.

If you compile for any transputer class other than TB, you must disable the compiler libraries with the compiler '**E**' option.

4.4.3 Error modes of compilation

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 the occam 2 compiler.

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.

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 UNDEFINED mode, is to ignore all run-time errors. This is potentially dangerous, but there are occasions when it is useful to avoid the run-time overhead of error checking, for example, where a program has already been proven correct. A second example is where results are being checked elsewhere.

The toolset compiler implements all three error modes, specified by command line options. The default is HALT system mode. All separately compiled units for a single processor must be compiled with the same error mode. Where a library is used the module of the appropriate error mode will be selected.

On the IMS T414, HALT mode does not work for processes running at high priority, as **HaltOnError** is cleared when going to high priority.

4.4.4 Mixing code with different error modes

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 allows the mixing of unchecked code (UNDEFINED) with code of other error modes, in a restricted manner.

To prevent accidental mixing of UNDEFINED code, an extra mode has been

added, called UNIVERSAL, specified by the compiler '**x**' option. UNIVERSAL code is the same as UNDEFINED, but has the property that it may be called from any other error mode. Code compiled in UNIVERSAL mode can only call code which is also in UNIVERSAL mode.

Note: UNIVERSAL mode is not intended as a general purpose facility and should be used with great caution, because it disables the security associated with error checking. It should only be used when error checking is not required and would be undesirable, such as with time critical code that is already proven.

Although the code produced in UNIVERSAL mode is the same as UNDEFINED mode it is important to distinguish between them. The behaviour of a system when an error occurs is not necessarily the same for both modes. This is because UNIVERSAL mode may be mixed with other modes, so an error occurring in UNIVERSAL code could halt the processor if it is mixed with HALT code. However, for the same code in UNDEFINED mode the behaviour at an error is not predictable.

Because the compiler libraries are only available in the HALT, STOP and UNDEFINED error modes you cannot use UNIVERSAL mode for any occam which requires the compiler libraries. When you compile any source in UNIVERSAL mode you should always disable the compiler libraries with the compiler '**e**' option.

4.4.5 Alias and usage checking

The compiler and syntax checker implement 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 communication, and that variables are not altered while being shared between parallel processes. For a further discussion of the rationale behind these rules, see *'INMOS technical note 32: Security aspects of occam 2'*.

Alias and usage checking during compilation may be disabled by means of compiler options. It is also 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, usage checking is automatically disabled.

Alias checking can impose some code penalties, for example, extra code is inserted if array accesses are made which cannot be checked until runtime. 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. No illegal programs, other than certain programs which use subscripted arrays with replicated **PAR**s, are accepted by the compiler. If a correct program is rejected because the compiler is imposing too strict a rule, it is possible to switch off the usage checker.

The syntax checker **icheck** provides better usage and alias checking than the compiler. If the checker passes a program that the compiler rejects then the checker can be assumed to be correct. In this case you can disable alias and/or usage checking in the compiler.

4.4.6 Using separate vector space

The compiler normally produces code which uses separate vector space. This means that the vectors declared within a compilation unit are allocated into a separate 'vector space' area of memory, rather than into workspace. 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. A compiler option disables the use of a separate vector space, in which case vectors are placed in the workspace.

When a program is loaded onto a transputer in a network, memory is allocated contiguously, as shown in figure 4.2.

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 a vector can be overridden by an allocation immediately after the declaration of an vector. 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 internal RAM. The program would be compiled with separate vector space enabled, but would include

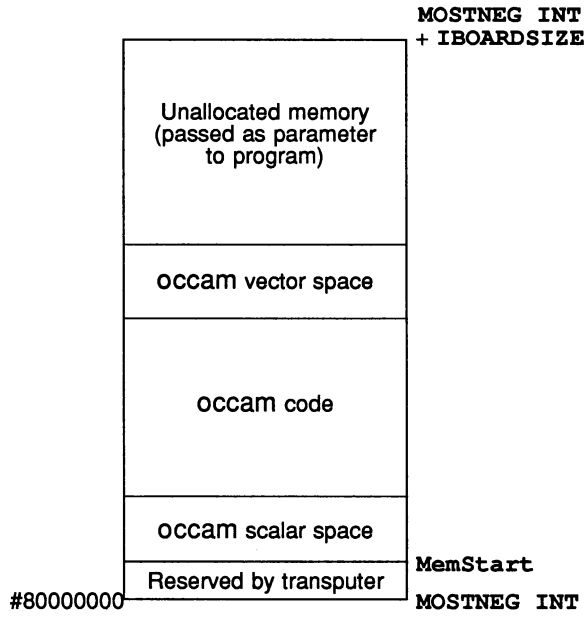


Figure 4.2 Memory allocation on a 32-bit transputer

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.

4.5 Sharing source between files

You can split the source of the program over any number of files by using the **#INCLUDE** directive. This directive allows you 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 ten. Files are treated according to the rules described in section 3.7.3. You may use any file name permitted by the rules in section 3.7.2. You should use **.inc** file extension 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.

4.6 Separate compilation

Separate compilation reflects the hierarchical structure of occam, and the occam compiler compiles occam procedures and/or functions (**PROC**s and **FUNCTION**s). 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 code file may have any name permitted by the rules in section 3.7.2. If you omit the extension the compiler assumes an extension in keeping with the current compilation, as described in section 3.7.2. Files are treated according to the rules described in section 3.7.3.

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

```
#USE "scunit.t4h"  -- code in file scunit.t4h
```

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 4.8.

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

4.6.1 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. Protocols 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.t4u"

  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 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**.

4.6.2 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 and then 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 or checked. 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 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.

4.7 Change control

When a change is made to part of a compiled 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 need to be recompiled, rather than needing to recompile the whole program. However, it would be tedious to have to remember which portions of a program had been edited in an editing session. For this reason a Makefile generator **imakef** is supplied with the toolset. 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 to do this it uses the file name rules described in sections 3.7.2 and 3.7.3.

See chapter 19 for details of the **imakef** tool and how to use it.

4.8 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 checking, 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 example:

```
#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. The file name of the library may be any name in accordance with the rules described in section 3.7.2. Files are treated by the rules in section 3.7.3.

4.8.1 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 checker and compiler are selective when a library is referenced. Only modules of a library that are of the same, or compatible, transputer type and error modes are read (see sections 4.4.2 and 4.4.4).

Selective loading is based on the following rules:

1 The transputer type 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 At least one routine (entry point) in a module is called by the code.

Rules 1 and 2 apply to the compiler and checker. All the rules are used by the linker. The compiler and checker only select on transputer type and error

mode. It is not until the linking stage that unused modules are rejected. For details on mixing processor classes and error modes see sections 4.4.2 and 4.4.4 respectively.

4.8.2 Building libraries

Libraries are built using the librarian tool **ilibr**. The librarian takes any number of separately compiled units (**.t***xx*), linked units (**.c***xx* files), or libraries (**.lib** 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 ensures that there are no multiply defined routines (entry points). In other words, for each combination of transputer type and error mode there may only be one routine with a particular name. In doing so the librarian treats a routine compiled for a transputer class as being equivalent to one copy for each member of that class. Routines compiled in universal error mode are treated as if there is one copy for each of the three error modes. These rules are described in more detail in chapter 16.

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, **mylib.t2h** and **mylib.t8h**. To form a library from these compilation units type:

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

The librarian uses the first file in the list to make up the output file name, so 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 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

ilibr -f mylib.lbb
```

When using the librarian you can specify the name of the library to be created using the 'O' option. For details of this and other options see chapter 16.

4.9 The pipeline sorter program

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

4.9.1 Overview of the program

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

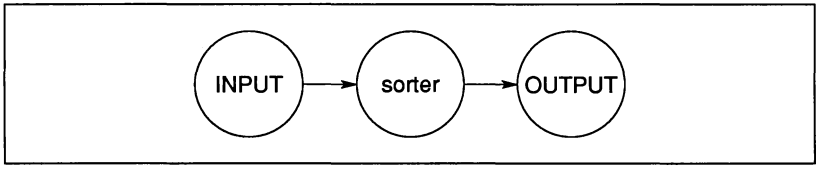


Figure 4.3 Basic structure of sorter program

Figure 4.3 shows the basic structure of this program. There are three processes: 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 4.4 shows the structure of the program using a pipeline.

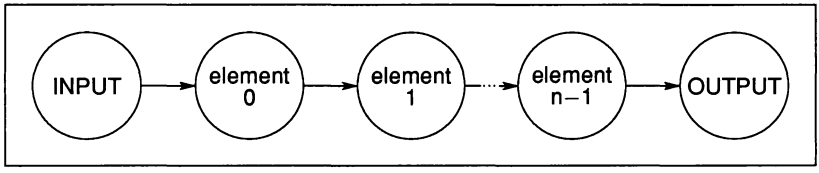


Figure 4.4 Pipeline of n elements

An obvious implementation would be to write an occam process for each process in figure 4.4, 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 (**PROC**s) 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 4.5 gives the final process diagram for the program.

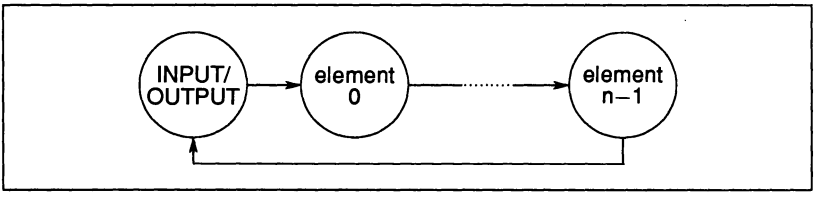


Figure 4.5 Program with combined input/output process

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

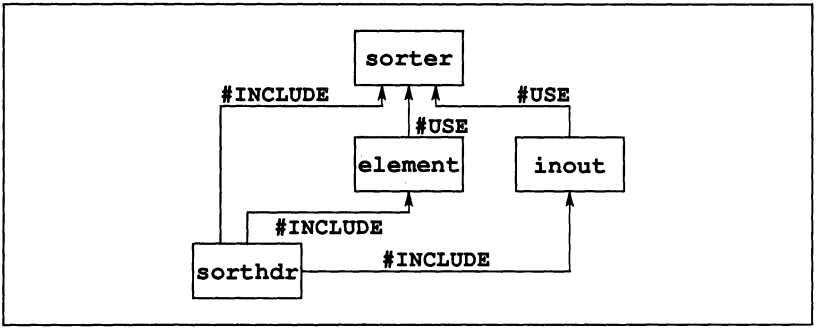


Figure 4.6 File structure of program

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.

4.9.2 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 no.processors IS 4:

VAL number.elements IS 100:
-- must be divisible by no.processors

VAL elements.per.processor IS
              number.elements / no.processors:
```

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.

The constants **no.processors** and **elements.per.processor** are not used by this version of the program, but are included for use by the multiprocessor version, described in section 5.6.

4.9.3 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 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.

4.9.4 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 to.pipe, from.pipe)

  #USE "hostio.lib"

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

  SEQ
    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
            SKIP
```

```
      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
:
```

4.9.5 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[0], pipe[number.elements])
    PAR i = 0 FOR number.elements
      sort.element(pipe[i], pipe[i + 1])
:
```

4.9.6 Building the program

To build the program, first check and 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 check the sorting element **element.occ** by typing:

```
icheck element
```

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

Next, compile the sorting element using one of the following commands:

```
occam element /e

occam element -e
```

The '**e**' option disables the compiler libraries and speeds up the compilation. The compiler libraries are not required because the program does not use any extended or floating point data types. 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 compiler produces a file called **element.t4h**.

Check the input/output process by typing:

```
icheck inout
```

Then compile the input/output process using one of the following commands:

```
occam inout /e
```

```
occam inout -e
```

The compiler will produce a file called **inout.t4h**.

Having checked and compiled the separate compilation units that make up the program you can now check the main body of the program. Type:

```
icheck sorter
```

Then compile the main body using one of the following commands:

```
occam sorter /e
```

```
occam sorter -e
```

The compiler will produce a file called **sorter.t4h**.

Having compiled all the components of the program you can now link them together to form a whole program. To do this type:

```
ilink sorter.t4h inout.t4h element.t4h hostio.lib
```

The file **sorter.t4h** *must* be the first file name on the linker's command line, because it is the main body, or entrypoint, of the program. The linker assumes that the first file in the input list contains the main entry point for the program and uses the transputer type and error mode of the first file to determine which libraries to select. The order of the remaining files is unimportant.

Any libraries used by the program must also be specified on the linker command line. The library **hostio.lib** is the server library used by this program.

The linker will create the files **sorter.c4h** and and **sorter.m4h**. The **.c4h** file contains the linked code, and the **.m4h** file contains a code map. The code map is used by the debugger and simulator tools.

Before you can run the program you must add bootstrap code. To do this use the bootstrap tool **iboot**, as follows:

```
iboot sorter.c4h
```

The bootstrap tool will create the files **sorter.b4h** and **sorter.d4h**. These **.b4h** file contains the bootable program code, and the **.d4h** file is a descriptor file for use by the debugger and simulator.

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

```
iserver /se /sb sorter.b4h

iserver -se -sb sorter.b4h
```

The program reads characters from the keyboard, sorts the line and redisplays it. The program will run until until input is terminated by typing RETURN on an empty line. The '**se**' option directs the server to terminate if the program sets the error flag.

To run the program using the simulator type:

```
isim sorter.b4h
```

When the simulator 'Monitor page' is displayed, type '**X**' followed by '**G**' to run the program.

4.9.7 Automated program building

It will be obvious from the previous section 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. If part of the program were changed then all affected parts must be recompiled, relinked and so on. To help manage these problems various software tools are available.

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.

In order for the Makefile generator to deduce compilation parameters and which tools to use you must give it the full name of the file you wish to build, including the correct file extension.

To generate a Makefile for the example program type:

```
imakef sorter.b4h
```

The Makefile generator will produce a Makefile for the program called **sorter**.

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. Because no extensions are given for the two compilation units **imakef** knows they are to be compiled for a T414 in HALT mode.

In order to compile the library file the Makefile generator searches for a library build file (**.lbb** extension) from which to deduce source file names and compilation details. As there is no such file for **hostio.lib** it assumes that the library is complete and cannot be rebuilt.

With this information about source file and their dependencies, it builds a Makefile containing full instructions on how to build the program.

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

For more details about the **imakef** tool, MAKE programs, and the format of Makefiles, see chapter 19.

5 Programming transputer networks

This chapter describes how to build programs that run on networks of transputers. It describes the steps in configuring a program and loading it on transputer network, and illustrates the procedures with an example program for four transputers. The chapter ends with a summary of program 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 runs on its own transputer and may communicate with other processes on other transputers via links.

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.

Figure 2.3 illustrates how occam processes can be run on a single transputer or configured on several transputers connected in a network.

To illustrate the use of the configurer, section 5.6 shows how the pipeline sorter program introduced in chapter 4 can be modified to run on a network of four transputers.

5.2 Configuration

Configuration is the mechanism by which processes to run on individual transputers in a network are collated with bootstrap code into a form which can be loaded onto a multitransputer network, and set running from the host computer.

The mapping of processes to transputers, channels to links and, implicitly, the interconnections between transputers are defined in the *configuration description.* Process placement and transputer connections are defined using special occam configuration statements. These are described in section 13.4.

Each processor in the network runs one or more separately compiled procedures. These separately compiled procedures may be considered as complete occam programs in their own right. The same separately compiled procedure can be

run on any number of processors; one copy exists in the configured code, and the code is loaded onto each transputer where the procedure runs.

The code for an individual processor is specified from within the configuration description using a **#USE** directive. Processes to run on individual transputers must be previously compiled and linked, and procedures to be used at configuration level may not be taken from a library.

5.3 Preparing for configuration

Before you can configure your network program you must first build the process(es) for each processor in the network. To do this you must compile all the components that will run on one processor and link them together to form one object file. This file must have the same name as that of the source of the outermost separate compilation procedure (**.occ** file), but with a **.c***xx* extension. All the code on one processor must be compiled for the same (or a compatible) transputer type and in the same (or a compatible) error mode, according to the rules described in sections 4.4.2 and 4.4.4.

As an example, suppose you have the source of a separate compilation procedure that you want to run on a T414 transputer in halt on error mode, in the file **node.occ**. It uses two separate compilation units, contained in the files **sc1.occ** and **sc2.occ**, and a library in the file **libs.lib**. Having compiled all the components link them together like this:

```
ilink node.t4h sc1.t4h sc2.t4h libs.lib
```

This links all the code which forms the processor's program and writes it in to the file **node.c4h**.

You should build the code for each transputer in a similar way.

Before using the configurer you must write a configuration description. This is done using the configuration statements of occam. The configuration description defines the arrangement of processors in the network and the allocation of code to each processor.

There must be at least one path from the host to all processors on the network so that the configurer can determine a load path for each processor. The load path is important because only one processor is connected directly to the host computer.

The code for each processor is referenced by a **#USE** directive. For example:

```
#USE "node.c4h" -- code is in the file node.c4h
```

In the configuration description you must explicitly declare the type of transputer that the code is to run on. This may specify a separate compilation procedure compiled for any compatible transputer type or class. For example if you have code to run on both a T414 and a T800 transputer you could compile it for the TA transputer class. You can then run that procedure on both the T414 and T800 transputers without the need to make separate copies of the procedure.

5.4 Configuring a program

The command **iconf** runs the configurer.

For example:

```
iconf program
```

As the default file extension for configuration descriptions is **.pgm**, the source of the description in this example would be in the file **program.pgm**.

When configuration is complete a new file, containing a bootable version of the code for the whole network, will have been created. This file has the same name as the description source, but with a **.btl** extension. In the above example it would be **program.btl**. A configuration description file with the **.dsc** extension is also created for the debugger.

See chapter 13 for details of the **iconf** tool and how to use it.

If you run the configurer with the '**M**' (configuration map) option, then it produces a readable configuration map file in place of a bootable network program. The configuration map shows details of processor interconnections and code allocation.

5.5 Loading a network

To load a network use the host file server in the same way as for single processor programs. For example:

```
iserver /sb program.btl

iserver -sb program.btl
```

A communication protocol exists between the root transputer and a target transputer network to direct the loading of code to the desired place in each transputer. The communication consists of bootstrap packets, routing information, address information, load information, code packets and execute items. For more infor-

mation on loading transputer networks see '*INMOS technical note 34: Loading transputer networks*'.

5.6 Example: A pipeline sorter on four transputers

This section describes how the pipeline sorter program, described in section 4.9, may be distributed over four T414 transputers.

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 5.1. 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.

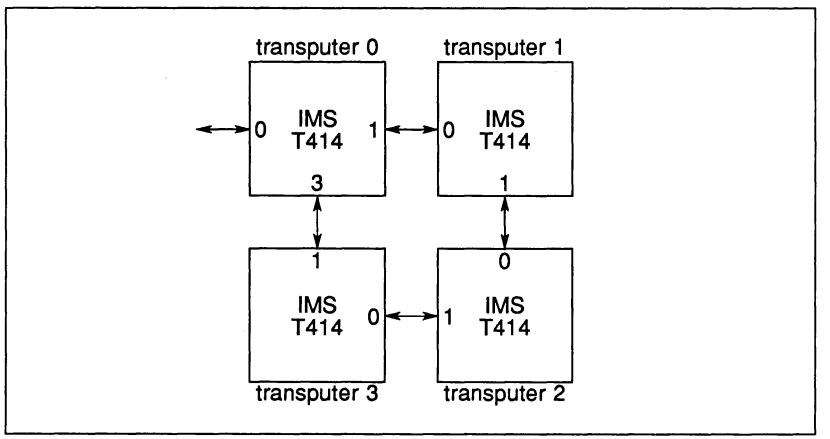


Figure 5.1 Network of four transputers

The occam source and configuration description developed in this example is supplied with the toolset 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 4.9.

The files are:

`sorthdr.inc`	the common protocol definition.
`element.occ`	the sorting element.
`inout.occ`	the interface to the host file server.
`tsort.occ`	part of the pipeline split to run on one transputer.
`sorter.pgm`	the configuration description for the network.

The contents of the files `sorthdr.inc`, `element.occ` and `inout.occ` are described in section 4.9. The contents of the other files used in the program are described below.

Three files are required to complete the program. These are the host file server library `hostio.lib`, the hostio include file `hostio.inc`, and the link address file `linkaddr.inc`. These files can be found in the toolset library directory.

5.6.1 The configuration description

This section describes how to build a configuration description, using the pipeline sorter program as an example.

First we must decide how the program will be mapped onto the transputers in the network. For the pipeline sorter we can distribute the processes as shown in figure 5.2.

For simplicity we can group together all the sorting elements that run on a single processor into one process called `tsort.occ`, which calls a sub-pipeline of sorter elements. We only need one copy of this process, even though it will run on more than one transputer. A copy of the code will be sent to each processor when the program is loaded.

```
#INCLUDE "sorthdr.inc"
PROC tsort (CHAN OF LETTERS pipe.in, pipe.out)

  #USE "element"  -- sorter element

  [elements.per.processor - 1]CHAN OF LETTERS pipe:

  PAR
    sort.element(pipe.in, pipe[0])
    PAR i = 0 FOR elements.per.processor - 2
      sort.element(pipe[i], pipe[i + 1])
    sort.element(pipe[elements.per.processor - 2],
                 pipe.out)
:
```

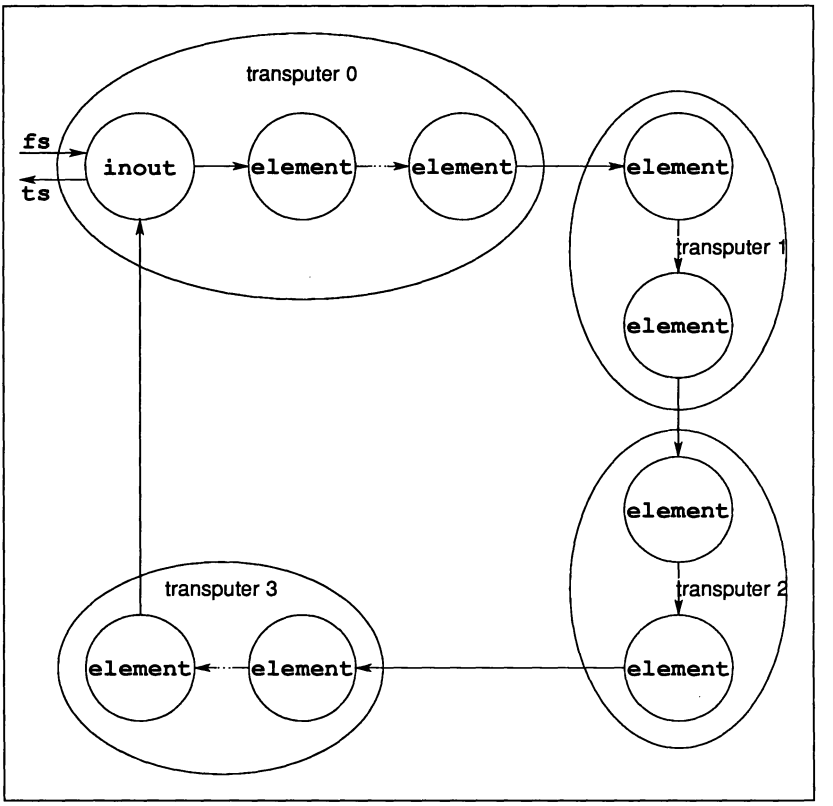


Figure 5.2 Pipeline sorter processes

Planning the configuration description

First draw a diagram of the network, label each link connecting processors, and allocate processes to each transputer in the network, as shown in figure 5.3. Use this diagram to draw up a table of process-to-transputer allocation like table 5.1.

Next, using figures 5.3 and 5.1, draw up another table show the mapping of channels to links, as in table 5.2. It can make the configuration description more concise and easier to write if you use an array of channels for channels that are to connect transputers together.

Now, using the tables, write a configuration description for the program. First reference any separately compiled units with the **#USE** directive and any con-

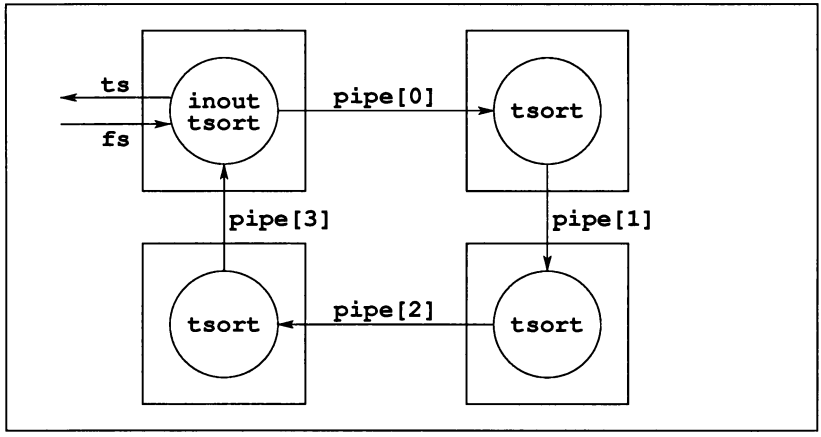


Figure 5.3 Allocation of processes on transputers

Process	Transputer	Type
`inout` and `tsort`	0	T414
`tsort`	1	T414
`tsort`	2	T414
`tsort`	3	T414

Table 5.1 Process – transputer allocation

	source		destination	
Channel	transputer	link	transputer	link
fs	*host*	–	0	0
ts	0	0	*host*	–
pipe[0]	1	0	0	1
pipe[1]	2	0	1	1
pipe[2]	3	0	2	1
pipe[3]	0	3	3	1

Table 5.2 Channel – link allocation

stant and protocol declarations with the `#INCLUDE` directive. Next declare the channels that are needed for inter-transputer communication. In this example the channels `fs`, `ts` and `pipe` need to be declared. Note that channels sourced on a transputer must be mapped to output links; destination channels are mapped to input links.

Next compile the input/output process using one of the following commands:

```
occam inout /e                    (creates file inout.t4h)

occam inout -e
```

For each transputer in the network you must:

- Place the necessary channels on links (from table 5.2).
- Call the process(es) to run on that transputer (from table 5.1).

To assign channels to links use the **PLACE** construct. Link numbers can be taken from table 5.2. Link addresses are defined in the include file **linkaddr.inc** supplied with the toolset.

To allocate code to processors in the configuration description, use the **PLACED PAR** construct. Processor numbers and types can be taken from table 5.1.

The basic configuration description is shown in figure 5.4.

To simplify the configuration description we can use a replicated **PLACED PAR** to place channels and allocate the **tsort** process to each of the four transputers. The final configuration description, which includes this refinement, can be found in the file **sorter.pgm** on the toolset examples directory. A listing of the program can be found in figure 5.5.

Note the use of the constant **no.processors** to make the description independent of the number of processors in the network.

We are now in a position to use the configurer. The next section describes the steps required to build the example program.

5.6.2 Building the program

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

```
occam element /e

occam element -e
```

Because the file has a **.occ** file extension you can omit the extension from the filename. The compiler option '**e**' is used because the compiler libraries are not required. The compiler will produce a file called **element.t4h**.

```
#USE "inout.c4h" -- linked inout process
#USE "tsort.c4h" -- linked sub-pipe process

#INCLUDE "hostio.inc"    -- host i/o constants
#INCLUDE "sorthdr.inc"   -- sorter constants
#INCLUDE "linkaddr.inc" -- link address constants

-- External channels to be mapped to links
CHAN OF SP fs, ts:
[4]CHAN OF LETTERS pipe:

PLACED PAR
  PROCESSOR 0 T414
    PLACE ts AT link0.out:
    PLACE fs AT link0.in:
    PLACE pipe[0] AT link1.out:
    PLACE pipe[3] AT link3.in:
    CHAN OF LETTERS to.start:
    -- For internal use on this processor
    PAR
      inout(fs, ts, to.start, pipe[3])
      tsort(to.start, pipe[0])

  PROCESSOR 1 T414
    PLACE pipe[0] AT link0.in:
    PLACE pipe[1] AT link1.out:
    tsort(pipe[0], pipe[1])

  PROCESSOR 2 T414
    PLACE pipe[1] AT link0.in:
    PLACE pipe[2] AT link1.out:
    tsort(pipe[1], pipe[2])

  PROCESSOR 3 T414
    PLACE pipe[2] AT link0.in:
    PLACE pipe[3] AT link1.out:
    tsort(pipe[2], pipe[3])
```

Figure 5.4 Example configuration

Now link it by typing:

```
ilink inout.t4h hostio.lib
```
(creates file `inout.c4h`)

```
#USE "inout.c4h" -- linked inout process
#USE "tsort.c4h" -- linked sub-pipe process

#INCLUDE "hostio.inc"    -- host i/o constants
#INCLUDE "sorthdr.inc"   -- sorter constants
#INCLUDE "linkaddr.inc" -- link address constants

-- External channels to be mapped to links
CHAN OF SP fs, ts:
[no.processors]CHAN OF LETTERS pipe:

PLACED PAR
  PROCESSOR 0 T414
    PLACE ts AT link0.out:
    PLACE fs AT link0.in:
    PLACE pipe[0] AT link1.out:
    PLACE pipe[no.processors - 1] AT link3.in:
    CHAN OF LETTERS to.start:
    -- For internal use on this processor
    PAR
      inout(fs, ts, to.start, pipe[no.processors - 1])
      tsort(to.start, pipe[0])

  PAR i = 1  FOR no.processors - 1
    PROCESSOR i T414
      PLACE pipe[i - 1] AT link0.in:
      PLACE pipe[i] AT link1.out:
      tsort(pipe[i - 1], pipe[i])
```

Figure 5.5 Final configuration

Now compile the sub-pipeline process using one of the following commands:

```
occam tsort /e
```
(creates file `tsort.t4h`)

```
occam tsort -e
```

Link it by typing:

```
ilink tsort.t4h element.t4h
```
(creates file `tsort.c4h`)

To make the program runnable you must add bootstrap code. To do this for multitransputer programs use the configurer `iconf`.

To configure the program type:

```
iconf sorter
```

The `.pgm` extension can be omitted because it is automatically assumed. The configurer will create the file `sorter.btl` and the configuration description file `sorter.dsc` for use by the debugger.

5.6.3 Running the program

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

```
iserver /se /sb sorter.btl

iserver -se -sb sorter.btl
```

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.

5.6.4 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.

To produce a Makefile for the entire program type:

```
imakef sorter.btl
```

The Makefile generator will produce a file called `sorter` containing a MAKE description for the program.

To build the program run the MAKE program on the file `sorter` and all the necessary compiling, linking and configuration will be done automatically. For more information about MAKE programs see chapter 19.

5.7 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) Allocating processes to transputers.

   (b) Allocating channels to links connecting processes on different transputers.

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 same name as the toplevel source file. The file is given a `.cxx` extension.

5 Run the configurer on the configuration description file.

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

Steps 3 to 5 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 explains how to use the program loading tools **iserver** and **iskip**. The chapter also discusses some aspects of debugging programs on transputer boards, describes how to load and debug programs that use the root transputer, and shows how to load a program onto a target network using the **iskip** tool.

6.1 Introduction

Programs to be run on transputers and transputer networks must consist of **bootable** code, that is, code to which bootstrap information has been added to make the program self-starting when it is loaded into the transputer's memory.

Bootable code is generated by **iboot** for single transputer programs and by **iconf** for multitransputer programs. The **iboot** tool generates a single bootable process to run on a single transputer, whereas **iconf** generates a bootable process for each transputer in the network. Bootable single transputer programs have the file extension **.b***xx*, and bootable multitransputer programs have either the extension **.btl** (boot from link programs) or **.btr** (programs that will be booted from ROM or dynamically loaded).

6.2 Tools for loading programs

Two tools are provided to assist with loading programs onto transputers and transputer networks:

- **iserver** – the loader tool and host file server. This tool loads the bootable code onto the transputer or transputer network and provides the runtime communications with the host system.

- **iskip** – the skip tool that allows a program to be loaded through the root transputer onto an external network. This tool starts up a special skip process on the root transputer that forces the root transputer to be skipped when **iserver** is invoked to load the program. The process consists of a software data link that transfers data byte by byte between the program and the host.

  Skip loading is useful for debugging programs that are configured to use the first transputer on a network. The root transputer in the network is

skipped and the program is loaded onto the first processor *after* the root transputer, leaving it free to run the debugger. This avoids having to use **idump** to save the root transputer's memory before invoking the debugger.

Programs loaded using **iskip** always require an extra processor. For example, a program written for a single transputer can only be loaded using **iskip** if there are at least two processors on the network, one to act as the root transputer and one to run the program.

6.2.1 The loading mechanism

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

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

6.3 Boards and sub-networks

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

Boot from link boards 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 that is self-starting when it is loaded into the transputer's memory.

Boot from link boards form the majority of boards in general use. Examples of boot from link boards supplied by INMOS are the IMS B014 VME motherboard, and the IMS B008 PC motherboard.

6.3.1 Boot from ROM boards

Boot from ROM boards are intended for applications such as embedded systems. Examples of boot from ROM boards are the IMS B002 single transputer board, and the IMS B006 multitransputer Double Eurocard. Boot from ROM boards can be set to boot from link by setting a single switch on the board.

Programs intended to run on boot from ROM boards contain no bootstrap code and are loaded down RS232 lines using special software. For details about the products available for installing transputer programs in ROM, including tools for programming EPROMS, contact INMOS.

All the example programs described in this manual must be loaded and run on boot from link boards or on boot from ROM boards that have been set to boot from link.

6.3.2 Subsystem wiring

Subsystem wiring is the way in which boards are connected together, and determines the manner in which transputer sub-networks 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 sub-networks.

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 by an external system to control the board. Down and Subsystem are output ports, used to propagate the Up signal out of the board in two different ways.

Down simply copies the Up signal, providing a way of propagating the signal unchanged throughout the network. Multiple boards can be chained together by connecting successive Up and Down ports and the whole network can be controlled by a single signal propagated from the root processor. Subsystem transfers control to the board, allowing sub-networks downstream of the board to be independently reset, analysed, and their error flags read, under the control of the root transputer.

6.3.3 Controlling sub-networks

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

In a typical system, the root transputer's Up port is connected to the host computer so that it can control the loading of programs and monitor for errors. The first processor in the sub-network 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 sub-network would normally be connected to Down on the root transputer. This would allow the host computer to reset the whole network with one 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 sub-network under the control of the root transputer. Here the sub-network would be connected to Subsystem so that the root transputer could repeatedly reset and re-load the sub-network. Any errors in the sub-network 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 always propagated through to the Subsystem port, but the error signal is not relayed back.

6.4 Debugging programs on transputer boards

When debugging transputer networks the debugger options that must be used vary with the board connections (subsystem wiring), whether or not the program uses the root transputer, and to some extent the board type.

6.4.1 Program mode

The debugger must execute on the root transputer of a network. This means that parts of the program that run on the root transputer must be debugged from a memory dump file. The file must be created by **idump** before the debugger is invoked, and the debugger must be invoked using a special option that allows it to read the dump file.

To avoid the need for a memory dump file, programs can be loaded onto the network over the root transputer by first invoking the skip tool **iskip**. This sets up a process on the root transputer that allows the rest of the network to communicate with the host as though the root transputer were absent. Since the root transputer runs no part of the program it can then be used to run the debugger without fear of overwriting the program.

For more details about loading and debugging programs that use the root transputer see section 6.4.3.

6.4.2 Board types

Some early boards such as the IMS B004, unlike later TRAM-based boards, do not propagate Reset through to the Subsystem port. On these boards the debugger must be invoked with different command line options than for the later boards.

Commands to use for specific board types are described in section 14.3.1.

6.4.3 Programs which use the root transputer

Programs that use the root processor to run part (or all) of the program can be loaded for debugging using **iserver** with or without **iskip**. If **iskip** is used an extra processor is required to act as the root transputer on the network.

If **iserver** is used the contents of the root transputer's memory must be dumped to disk using the **idump** tool, and the debugger invoked with the '**R**' option to read the memory dump file. The rest of the network can be debugged directly down the transputer links. If you only have a single processor this method *must* be used.

If **iskip** is used, the program can be debugged down the transputer link in the normal way. The skip tool allows the program to be loaded into the sub-network over the top of the root processor, leaving it free to run the debugger. Communication with the host is supported by a special route-through mechanism running on the root transputer that transfers data transparently between the program and the host computer.

6.4.4 Programs which do not use the root transputer

To load a program for debugging that does not use the root transputer, first ensure that the program is configured for the correct sub-network so that it does not include the root processor, invoke **iskip** to set up the skip process on the root transputer, and load the program in the normal way using **iserver**. To debug the program invoke the debugger with the '**T**' option and specify the root transputer link to which the network is connected.

6.4.5 Analyse and Reset

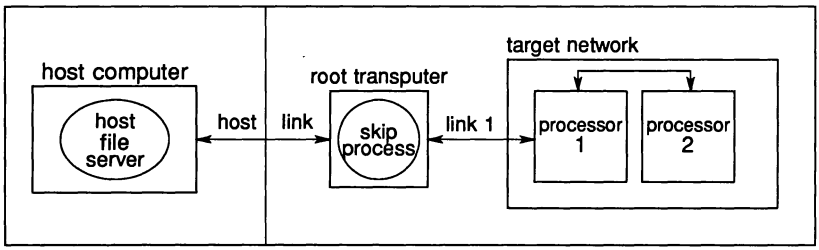
Care must be taken that **Analyse** or **Reset** are only asserted once on a network that is to be debugged, or incorrect data will be obtained. To prevent this the debugger should be invoked using the command sequences given in table 14.2.

6.5 Example of using iskip

This example shows how to use the **iskip** tool to load a program over the root transputer. The program to be loaded is configured for a two interconnected T800 transputers mounted on a B008 motherboard. This forms the target network. A T414 processor in slot zero acts as the root transputer, and the target network

is connected to link 1 on the root transputer via one of the links on processor 1.

The target network and its connections to the host computer are shown schematically below.



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

To prepare the board for running the program on the two-processor T800 network, invoke **iskip** using one of the following commands:

```
iskip 1 /r /e

iskip 1 -r -e
```

This sets up the system to direct the program to the target network and activates the data transfer program 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

iserver -ss -se -sc twinprog.btl
```

7 Debugging occam programs

This chapter describes how to debug occam programs. It begins with an introduction to the toolset debugger, describes symbolic and assembly level debugging, and illustrates the facilities available with a tutorial example. The chapter also describes the simulator tool that allows you to debug programs without hardware, and contains some hints for debugging occam programs.

7.1 Introduction

The debugger tool **idebug** provides an interactive environment for the post-mortem debugging of occam programs. It allows processes running on each transputer on a network to be examined at source code and assembly code level. It supports the inspection of source level symbols such as variables, channels, and timers, and the disassembly and inspection of memory, for any process running on any transputer in the network.

The toolset debugger allows limited symbolic debugging of other languages. Details of these facilities can be found in section 7.2.2.

7.1.1 Compiling programs for debugging

Programs to be debugged must be compiled with debugging data enabled. This is the compiler default and can only be disabled by specifying the compiler '**D**' option. When debugging is enabled, the compiler makes no change to the object code, but rather inserts extra information into the object file for the debugger to use.

Programs to be debugged should also be compiled in HALT mode. In HALT mode any error during program execution halts the transputer immediately. In STOP and UNDEFINED modes, errors do not halt the program, and programs compiled in these modes can only be debugged if they are halted explicitly. HALT mode is the compiler default.

A running occam program may halt for a number of reasons. Examples of these 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. Alias checking is normally performed by the compiler, but can be disabled.

For a full list of possible causes of run time errors, see section 14.5.2.

When a run time 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.

Note: The debugger may not find all current processes; for example, it cannot find processes which have deadlocked waiting for communication. Deadlocks are discussed in more detail in section 7.4.

7.1.2 Programs that can be debugged

The debugger can analyse programs running on transputers that are either directly attached to a host through a server program, or connected to the host via a root transputer. The debugger runs on the root transputer and networks to be debugged must incorporate a 32-bit transputer at the root.

For programs that use the root transputer in a network, the contents of the transputer's memory must be dumped into a special memory dump file for the debugger to read. This is because the debugger itself runs on the root transputer and overwrites its memory.

For programs that do not use the root transputer, it is not necessary to dump the root transputer's memory. The root transputer does not run any part of the program and can be used to run the debugger.

Programs that do not use the root transputer can be loaded onto the network using the skip loader program **`iskip`** that loads *over* the root transputer and leaves it free to run the debugger. The debugger accesses all other transputers on the network through the transputer links. For more information about skip loading see chapter 6.

7.2 Debugger facilities

The debugger facilities divide into two groups; those that operate at the level of the high level language, (symbolic facilities), those that operate at assembly code level (Monitor page facilities).

Symbolic facilities allow programs to be debugged from source code. Source code can be scanned, variables inspected, and procedure calls back traced. Symbolic functions can also be used to debug non-occam programs, with some restrictions.

Symbolic functions are invoked using function keys. For details of specific keyboard layouts see the Delivery Manual.

The Monitor page facilities view the transputer network from the assembly code level, and do not use the debugging information produced by the compiler. They give information about the transputer's state, such as the status of process and timer queues. Either set of facilities may be used on any transputer in the network.

7.2.1 Symbolic debugging

Symbolic debugging operates with the symbols that are defined in the high-level language source code. The debugger symbolic interface allows you to locate specific source code, inspect variables and channels, and trace procedure calls.

Symbolic facilities for non-occam languages are more limited; for details see section 7.2.2.

Locating to source code

Given any transputer instruction address, the debugger can locate the corresponding occam line in the source file and display it. This is known as *locating to* the code.

The debugger can display the occam source line corresponding to any of the following:

- The last transputer instruction executed.
- A process running in parallel.
- A process waiting for a timer.
- A process waiting for communication on a transputer link.

A process waiting for communication on an internal channel can be found by inspecting the contents of that channel, as described later in this section.

The ability to locate to any source line requires the source to be available. Where the source is not available, for example for the toolset compiler libraries, the

debugger can only locate to the *call* to the routine, rather than the source line itself.

When a source line has been located, other debugger functions can be used to examine the code and inspect program parameters. These functions are accessed through special function keys. For a full list of these functions, see chapter 14.

Examining code and inspecting parameters

Once a source line has been located, parameters that are in scope at that line can be examined using the INSPECT function, and procedure calls can be located using the BACKTRACE function. Values of constants, variables, parameters, abbreviations, array elements, and channels, can be all be inspected, and non-local variables and channels can also be inspected. They can be displayed in hexadecimal, or in any other valid occam representation for their type.

From any location it is possible to backtrace from a procedure or function call to the point from which the enclosing procedure is called. This works even if the source is not present, as may be the case with libraries supplied from external sources. The process can be repeated for each nested procedure or function call, to build up a complete stack trace of procedure calls from a particular location. Variables and other parameters can be inspected at any stage.

Using CHANNEL it is possible to determine the instruction and workspace pointers of any process waiting for communication, jump to the waiting process, and continue debugging at that point.

Other functions that are available during symbolic debugging are the examination of specific symbols, and the determination of address and workspace requirements for procedures and functions.

7.2.2 Debugging non-occam programs

The debugger can analyse non-occam programs using information supplied by the compilers. This may be limited and can restrict the ability of `idebug` to debug non-occam code. For C, FORTRAN, and Pascal programs the BACKTRACE function can be used to discover the context of procedure and function calls, but the INSPECT and CHANNEL functions are inoperative. However, other types of debugging operations, such as locating waiting processes, can still be performed.

Note: Non-occam languages can in some circumstances set the error flag even during correct execution and as a result of this non-occam programs compiled in HALT mode may produce spurious errors.

7.2.3 Assembly level debugging

The debugger can also debug programs at the level of assembly code and transputer state, using the Monitor page facility. To enter the Monitor page debugging environment, invoke the MONITOR function from the symbolic debugging environment.

From the Monitor page environment you can inspect the instruction pointer (program counter), the workspace descriptor, the process queues, the processor flags and the transputer memory contents.

Using information from the process and timer queues, the debugger can display

- Instruction and workspace pointers of queued processes
- Processes waiting for communication on transputer links
- Processes waiting for a signal on the **Event** pin.

Memory can be displayed in ASCII, hexadecimal, as any occam type, or by disassembly into transputer instructions. The disassembly translates memory contents directly into transputer instructions; it does not insert labels, nor provide symbolic operands. Assembly level debugging can also provide a *memory map* showing the positions of code and workspace for each transputer in the network.

7.3 A debugging example

This section describes a debugging session for an example program. It shows you how to debug source code using the symbolic functions, and how to examine object code using the Monitor page commands.

For details of the debugger and descriptions of its options, see chapter 14. Chapter 14 also describes debugger functions that are not described here, such as the inspection of arrays and multiprocessor debugging.

The source of the example program is provided in the file **debugex.occ** which can be found in the toolset 'examples' directory.

The example can be run on any transputer board with at least 2 Mbytes of memory.

7.3.1 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 some debugging methods.

The program uses five processes, each coded as a separate procedure. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result.

The example program is listed below.

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

PROC debug.example (CHAN OF SP fs, ts,
                    []INT free.memory)

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

  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 (CHAN OF INT in, out)
    INT n :
    SEQ
      in ? n

      SEQ i = 0 FOR n
        out ! i

      out ! stop.integer
  :
```

```
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 (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 (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 (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.newline      (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)
:
```

7.3.2 Building a loadable program

To compile the program use the following sequence of commands:

```
occam debugex

ilink debugex.t4h hostio.lib convert.lib occambh.lib

iboot debugex.c4h
```

This sequence compiles the program in HALT mode for a T414 transputer (by default), links in the libraries that are used in the program, and adds bootstrap code. If you are using a T425 or T800 transputer, remember to specify the transputer type when you run the compiler, (**T5** or **T425** for the T425, **T8** or **T800** for the T800). For the T800 processor you must use the appropriate file extensions and link in the correct compiler library. For a list of the compiler libraries see section 24.2.

The end product of the compilation, linking and booting operations is the bootable file **debugex.b4h** (or equivalent for T425 and T800 transputers) that can be loaded onto a single transputer.

The program can also be built using **imakef** and a suitable MAKE program. First create a Makefile by running the **imakef** tool as follows:

```
imakef debugex.b4h.
```

This creates a Makefile called **debugex** which contains details of how to build the program to run on an IMS T414 in **HALT** error mode.

If you are using an IMS T425 or IMS T800 transputer, use the appropriate command:

```
imakef debugex.b5h

imakef debugex.b8h
```

You can then produce an executable code file by running MAKE on the resulting Makefile, as in the following examples:

```
make -f debugex
```

For more details about using the **imakef** tool see chapter 19.

7.3.3 Host environment variables

Before running the program check that the host environment variables **ITERM**, **IBOARDSIZE**, and **TRANSPUTER** are set up on the system. For more information about setting up environment variables on your system see the Delivery Manual.

7.3.4 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 debugex.b4h

iserver -se -sb debugex.b4h
```

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- transputer error flag has been set.
```

7.3.5 Creating a memory dump file

To create a memory dump file for the debugger to read, type:

```
idump debugex 10000
```

This creates a file called **debugex.dmp** containing the transputer's register contents and the first 10000 bytes of memory. You are then returned to the operating system prompt.

7.3.6 Running the debugger

To debug the example program, use one of the following commands:

```
idebug debugex.b4h /r debugex

idebug debugex.b4h -r debugex
```

The '**r**' option identifies the program as one that was executed on the *root* transputer and specifies the memory dump file to be read.

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
#80000360)
```

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:#80000611 and Wdesc:#80000285
```

```
(Lo) (at #800004B0)
```

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 to the procedure **sum**, and thence to the procedure **square**. While 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.
LINKS	Displays a list of links to other transputers. Useful when debugging multitransputer programs.
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 another source 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.

7.4 Hints for debugging occam programs

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 part 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.

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. If a default case is not given, the debugger can only locate to the line *following* the construct.

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 procedure:

```
PROC deadlock ()
  CHAN OF INT c :
  PAR
    SEQ
      c ! 99
      c ! 101

    INT x :
    c ? x
:
```

The program can be debugged by adding a process that will remain idle while the program is debugged. An example of the type of code that is required is illustrated below.

```
PROC deadlock.debug ()
  CHAN OF INT c :
  CHAN OF INT stopper :
  PAR
    VAL one.second   IS 15625 :  -- Low priority
    VAL secs.per.day IS (60 * 60) * 24 :
    VAL one.day      IS one.second * secs.per.day :
    TIMER time :
    INT now :
    SEQ
      time ? now
      ALT
        time ? AFTER now PLUS
                  one.day -- will locate to here
          SKIP
        stopper ? now
          SKIP

    SEQ
      PAR
        SEQ
          c ! 99
          c ! 101           -- will jump to here

        INT x :
        c ? x
      stopper ! 0
:
```

The new process uses a TIMER within an **ALT** statement. Similar code is sup-

plied in the process library as the procedure **debug.timer**.

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, 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 **ALT** statement.

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 toolset not to alter the run time 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.

7.5 Debugging using embedded messages

Transputer programs can also 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.

7.5.1 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.

7.6 Notes on using the debugger

Invalid pointers

The debugger checks instruction pointers and workspace descriptors for the correct code and data limits. Invalid pointers are flagged by an asterisk ('`*`') on the debugger screen display.

Invalid pointers indicate a major problem with the program. They are also caused by specifying the wrong dump file.

Locating within the ALT construct

The debugger is unable to locate to specific alternatives within ALT constructs. If a channel is waiting within an ALT, the debugger can only locate to the *first* alternative in the list, no matter which channel or timer is requested.

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.

```
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
:
```

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**.

7.7 Debugging with the T414 simulator

The T414 simulator provides an interactive simulation of a single IMS T414 transputer, running on a 2 Mbyte boot from link transputer board, and connected to a host computer through the host file server **iserver**. The simulator allows any single processor program compiled for the T414 to be run using exactly the same code as would be loaded onto a transputer evaluation board.

All the component parts of a program to be simulated, including its libraries, must be linked together using **ilink**, and made bootable using **iboot**. The file extension of a program being simulated must therefore be of the **.b4***x* type.

Like the debugger, the simulator has two command interfaces. The first is the Monitor page which allows low level features of the transputer and the program to be accessed. The second is the symbolic interface, which allows programs to be debugged in terms of occam source code.

Both interfaces provide the same features as are provided by the debugger, with the addition of execution monitoring commands [SET BREAK], [SINGLE STEP] and [WALK] on the symbolic interface.

For details of the **isim** command and its options see chapter 21.

Note: The T414 simulator can only be used to run single transputer programs.

7.7.1 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 and memory can be examined directly at the Monitor page, and source code can be examined using the symbolic interface.
- To monitor the execution of programs using single step execution and the setting of break points within source code or at specific memory locations. At the Monitor page level code can be executed instruction by instruction, and at the symbolic level code can be executed line by line.

7.7.2 Standard debugging

The simulator provides all the symbolic and low level debugging facilties of the toolset debugger. The facilities are only described briefly here. For more details see chapters 14 and 21.

Symbolic facilities

The simulator symbolic interface allows the source of any occam program to be examined. The values of constants, variables, parameters, abbreviations and arrays can be inspected using special function keys.

To use the simulator symbolic features, the program must be compiled with debugging data enabled, that is, without the compiler 'D' option. When compiling a program for debugging it is best to compile it in HALT mode.

Low level facilities

Using the Monitor page facilities, the simulator can display the simulated transputer's internal state, showing the instruction pointer (`Iptr`), the workspace descriptor (`Wdesc`), the status of process timer queues, the contents of registers, and the values of status flags.

Memory can be displayed in ASCII, hexadecimal or any other occam type, or disassembled into transputer instructions.

7.7.3 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:

- Break points
- Single step execution of a program
- Single step execution of a single process (walking)
- Modifying registers and memory locations

Break points

Break points can be set at the occam source level by moving the cursor to the source line and pressing [SET BREAK]. The break point is set at the beginning of that line. When a break point is encountered while the program is running, execution is halted and the line containing the break point is displayed. At this point the line has not been executed.

Break points can also be used at the Monitor page. They can be set, displayed, and cancelled using the 'B' command to display the Breakpoint Options Page.

Single step execution

There are two types of single stepping at the source level; [SINGLE STEP] which skips to the next process if the process deschedules, and [WALK] which remains in the process even if an error occurs.

Both [SINGLE STEP] and [WALK] execute a single line of code.

At the Monitor page a program can be stepped a single instruction at a time using the 'S' command.

Changing registers

When a program halts the value of any register can be changed. However, the effect on the program of changing registers may be unpredictable and the practice is not recommended.

7.8 Simulator example

This section shows how to use the simulator to debug programs by interactively monitoring execution of the program. The program used to demonstrate this use of the simulator is the example program used earlier in this chapter.

7.8.1 Running the simulation

Before starting the simulation you must create a bootable version of the program. The simulator requires a bootable file compiled in HALT mode for a T414 processor.

Start the simulation by typing:

```
isim debugex.b4h
```

A Monitor page display appears on the screen. To obtain more information about the Monitor page commands, press '**?**'.

Like the debugger, the simulator uses two pointers to identify the code to be examined: the instruction pointer **Iptr** and the workspace descriptor **Wdesc**. When the simulator starts **Iptr** points to the start of the bootstrap code for the program. To start the program running under the simulator, press '**X**'. This loads and boots the program and leaves the pointers at the start of the program.

First use the 'O' command to examine the occam source. Press 'O' to invoke the command, and then press [RETURN] twice to accept the current pointer values. The simulator then displays the first line of the program. Press [MONITOR] to return to the Monitor page.

The program can be run either from the source level by pressing [GO], or at the Monitor page by pressing 'G'. If the program fails symbolic or Monitor page facilities can be used to trace the fault.

Setting break points

Break points can be set at any point in the program. For example, suppose you suspect that procedure **square** contains an error. To examine its behaviour you can set a break point at the procedure and execute it line by line.

Before break points can be set the debug information for the module must be loaded, using the Monitor page 'C' command. When you invoke the command, you are prompted for the module name, which in this case is **debugex** (the name of the file containing the program). When the debug data is loaded, you can use the 'P' command to display procedure and function names.

To set a break point use the 'B' command. This displays an options menu called the Breakpoint Options Page. Choose option 2 – **Set break point at procedure** – and type the name **square** followed by RETURN. To confirm that the break point has been set correctly, return to the Breakpoint Options Page and invoke option 3 – **Display breakpoints**.

To cancel the break point use option 4 or option 5 on the Breakpoint Options Page.

Break points can also be set and cancelled within occam source. The procedure is described in section 7.8.4.

7.8.2 Starting the program

Start the program executing using the Monitor page command 'G'. The simulator immediately prompts for a break point address, to which you can respond by typing RETURN. This facility of interactive break point setting differs from the normal method in that the break point is cancelled once it is reached. For the purpose of this tutorial you can ignore this facility.

The program runs until it encounters the break point in the procedure, and then returns to the Monitor page display. The line:

```
Sum of the first n squares of factorials
```

is displayed at the bottom of the screen. To remove the line from the screen display, press REFRESH. On the Monitor page **Iptr** and **Wdesc** now displays new values.

To examine the occam source, use the 'O' command and press RETURN twice to accept the current program pointers. The cursor moves to the start of procedure **square**, and you can either use symbolic debugging to examine the source, or single step the code as described in the following section.

7.8.3 Single step execution

To examine the behaviour of procedure **square** you can execute the code line by line. At any point in the code you can use symbolic functions such as INSPECT to examine variables.

Having located to procedure **square**, press STEP. This moves the cursor to the first executable line in the procedure, namely:

```
in ? x
```

Examine the channel by placing the cursor on the word **in**, and press INSPECT. The channel is empty, indicating that nothing is being sent on the channel, and the input procedure cannot proceed until the other process is ready to communicate. Now press STEP again and the cursor moves to the line containing the call to procedure **facs**, which is the next process waiting on the queue. Because of the delay in the previous process, this process has started to communicate on the channel.

Press STEP again to move the cursor to the first line in procedure **facs**. Press STEP once more to move the first executable line in the process, position the cursor over the channel **in**, and press INSPECT. This channel too will be empty.

To examine procedure **facs** further and avoid jumping to another process, use WALK in place of STEP. This steps a single line of occam source but does not jump to other processes as they become scheduled.

Press WALK successively to to step through the **facs** process. When a prompt appears, type a response as normal and use REFRESH to clear the display. As you step though the code of the procedure, you will enter the function **factorial**, returning eventually to the body of **facs**.

7.8.4 Setting break points in source

To return to the procedure **square** while single stepping the program set a break point in the occam source. Move the cursor to the line:

```
sq := x * x
```

and press SET BREAK. This sets a break point at that line.

Press GO to start the program running. The program runs until the new break point is reached. You can now use symbolic debugging to inspect values within procedure **square**, or continue to single step or walk the program starting at the break point.

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 appendix F.

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 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 chapter 24.

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:

0 standard input
1 standard output
2 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 pressing the system interrupt key.

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.

The following code places the host channels correctly on the root processor:

```
#INCLUDE "linkaddr.inc"
#INCLUDE "hostio.inc"
CHAN OF SP fs, ts :
PLACE fs AT link0.in :
PLACE ts AT link0.out :
```

In this example **hostio.inc** contains the **SP** protocol definition, and **linkaddr.inc** contains the transputer link addresses. Both files are provided with the toolset.

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.

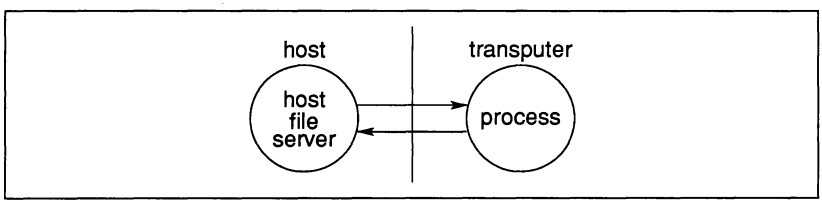


Figure 8.1 Program input/output

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 section 24.4.9.

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.

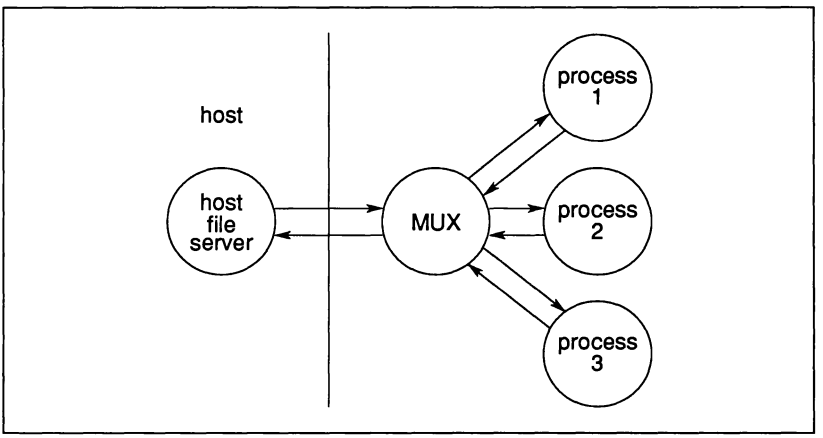


Figure 8.2 Multiplexing the host file server

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.

```
#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
    [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

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.

9 Mixed language programming

This chapter describes how to build programs in a mixture of programming languages. It describes how to use the occam toolset and the C, FORTRAN and Pascal compilers supplied by INMOS to compile mixed language programs, and run them on networks of transputers.

The chapter is structured as follows:

Section 9.1 introduces mixed language programming for transputers, and describes the C, FORTRAN and Pascal code interfaces to occam.

Section 9.2 describes the equivalent occam process and some aspects of its use. It explains how routines and programs written in other languages are integrated into occam code by making them appear like separately compiled occam code.

Sections 9.3, 9.4, and 9.5 describe the three interfaces through which other language processes are called from occam and also describe the language-specific routines that implement channel communication.

Section 9.6 describes the communication library routines for the three languages.

Section 9.7 explains how to call occam from C, FORTRAN and Pascal.

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 processes, communicating via channels, which can be distributed in any way to a network of transputers.

Programs written in C, FORTRAN, and Pascal are treated as independent processes and like occam processes can be run on any transputer in the network. Interfaces between occam and C, FORTRAN, and Pascal programs are clear and well-defined, making it easy to use the languages in parallel programs. Programs written in these languages must conform to one of several formal

procedure definitions.

INMOS supplies toolset-compatible compilers for C, FORTRAN, and Pascal. Special library routines to support channel input and output are supplied with the compilers.

Mixed language programs can be debugged using the toolset debugger **idebug**, with some restrictions. For more information see section 14.4.3.

It is also possible to call separately compiled occam procedures from C, FORTRAN and Pascal. 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 The equivalent occam process

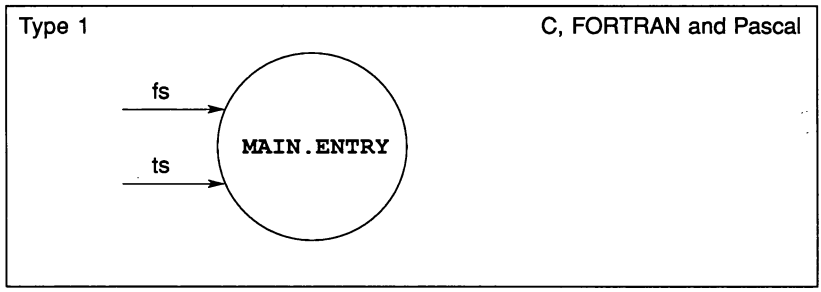
The occam programming model consists of independent processes running in parallel with other processes and communicating via channels. In accordance with this model, programs written in C, FORTRAN and Pascal are viewed as single, separately compiled processes that can be run in parallel with each other, and with other occam processes.

To form an equivalent occam process for a C, FORTRAN or Pascal program, the object modules that make up the program (including run time libraries if used) are linked with special occam interface code using **ilink**. This produces a *.cxx* file.

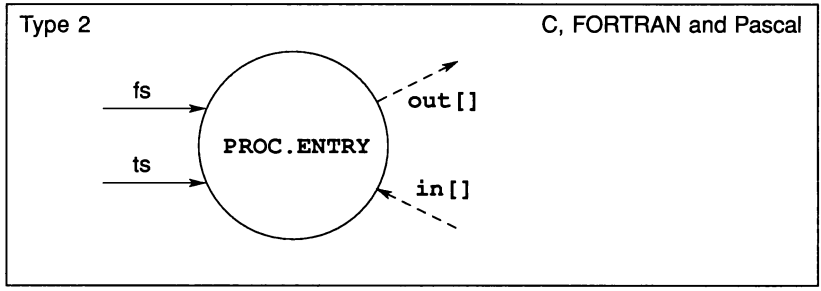
9.2.1 occam interface code

occam interface code provides a fixed interface between occam and C, FORTRAN or Pascal programs. There are three types of interface code, known as types 1, 2, and 3. Descriptions and process diagrams for the three interfaces follow.

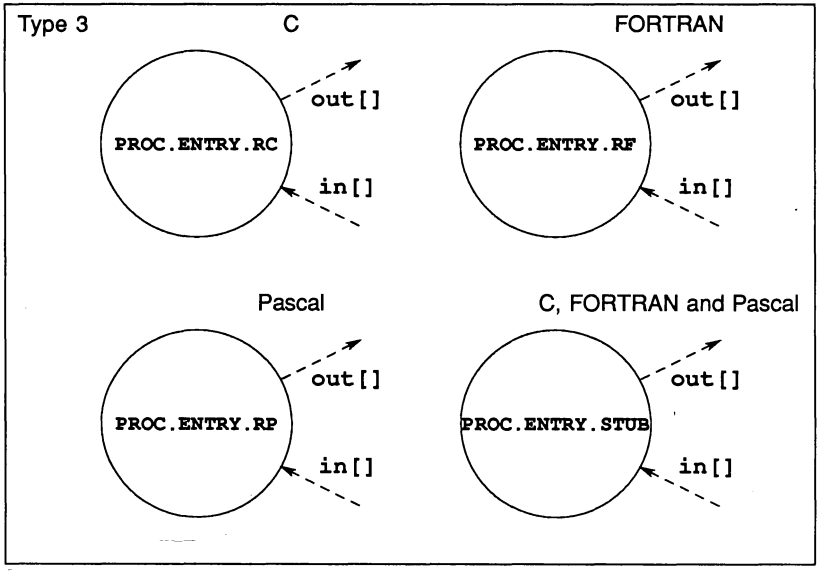
Type 1 : This interface is used when the program runs on a single transputer and communicates only with the host file server. It can be used for C, FORTRAN and Pascal programs.



Type 2 : This interface is used when the program communicates with other processes as well as the host file server. This interface can be used with C, FORTRAN, and Pascal programs and is used with the full versions of the respective run time libraries.



Type 3 : This interface is similar to the type 2 interface except that there is no access to the host file server. The interface is used with the reduced version of the run time library, which contains only startup, maths, string and channel i/o routines and does not include the routines to access the host i/o services (for example, access to files and the console). The type 3 interface exists in several forms for use with C, FORTRAN, and Pascal programs, as illustrated below.



Type 2 and 3 interfaces are called from the enclosing occam code and may be a part of a network of occam processes.

To use an equivalent occam process in occam source, first declare the process within the occam program, using the **#IMPORT** directive. For example, to use the equivalent occam process defined in the file **proc1.c8u** (which will be the output of the linker after a C, FORTRAN or Pascal program has been linked with appropriate occam interface code), include the following line in the occam source:

```
#IMPORT "proc1.c8u" -- contains C program cproc1
```

See section 23.6.4 for a detailed description of the **#IMPORT** directive.

The **#IMPORT** directive for equivalent occam processes has an identical effect to the **#USE** directive; in both cases the compiler reads the file and extracts the necessary information to make the call to the equivalent occam process. **#IMPORT** is used rather than **#USE** to reference equivalent occam processes in order that the Makefile generator tool **imakef** can distinguish them from separately compiled occam processes.

Equivalent occam processes can be collated into a single library file using the librarian tool **ilibr**.

9.2.2 Reserved channels

All equivalent occam processes have four reserved channels, namely `in[0]`, `in[1]`, `out[0]` and `out[1]`. No process which uses host services through the full run time library should use these channels.

The first two elements of both vectors of channel pointers are reserved as follows:

`out[0]`	Reserved for diagnostic output.
`in[0]`	Reserved for diagnostic input, but not currently used.
`out[1]`	Commands and data from the run time library to the host file server.
`in[1]`	Responses from the host file server to the run time library.

9.2.3 Error modes

When linked with occam, C, FORTRAN and Pascal programs take the same compilation error mode as the occam interface code used in linking the program. The `.t`*xx* and `.c`*xx* extensions of the interface code files indicate the transputer type and error mode of the code, according to the rules described in section 3.7.2.

Note: If the network includes any processes written in C then you are recommended to use only UNDEFINED or UNIVERSAL error modes throughout the system. For Pascal and FORTRAN you can use HALT, UNDEFINED or UNIVERSAL mode.

9.2.4 Stack and heap requirements

Data storage (work space) requirements for C, FORTRAN, and Pascal programs are set according to the values of `flag`, `ws1`, and `ws2`. Work space is allocated by the language compiler and run time libraries.

Stack, static data and heap requirements vary from program to program, and between languages. The work space vectors passed to the program must be large enough to accommodate:

- The stack the program needs when it runs.
- All the static data required by the program.
- The heap used by the program and the run time libraries.

Stack overflow may lead to unpredictable behaviour by the program. For these reasons it is best to run a program, at first, with a large combined stack and heap.

Later, when you have run the program and determined stack and heap usage, you may use a separate stack and heap, tailored to your application. Separate work spaces allow you to ensure that the stack is resident in the transputer's internal memory, and enables the program to run faster. Procedures and methods you can use to optimise stacks are described in *'INMOS technical note 17: Performance maximisation'* and *INMOS technical note 55: Using the occam toolsets with non-occam applications.*

A minimum stack size of 512 words is recommended.

Stack overflow

Failure or unpredictable behaviour of programs may be due to stack overflow; to test for this in a program, use the procedure outlined below.

1 Initialise the bottom few words of the stack (a falling stack is used) to some pattern of values.

2 Run the program and, after it crashes, use the debugger to examine the values in the stack. If the values you initialised have been changed then stack overflow is likely.

3 Increase the stack size and try again.

The same method can be used to determine static data and heap requirements, except that these use a rising stack.

The following occam fragment gives an example of initialising the bottom of the stack:

```
SEQ i = 0 FOR words.to.initialise
  ws1[i] := i
```

9.3 Type 1 interface

The type 1 interface is used when making a program to run on a single transputer, which does not communicate with any other process apart from the host file server.

C, FORTRAN or Pascal programs that run on a single transputer do not need to use occam. Compile the program as usual and then link in the type 1 occam interface code, using **ilink**. This builds the equivalent occam process for the program. It makes the program appear like occam and enables you to add normal bootstrap code.

The code for the type 1 process is essentially the same as for any occam process for a single transputer, except that an extra parameter is required for the C, FORTRAN, or Pascal program's run time stack. This only applies if a separate non-occam stack was requested when **iboot** was invoked. The size of the stack is determined by the parameter supplied with the **iboot** '**S**' option.

9.3.1 Type 1 procedural interface

The type 1 occam interface is defined as follows:

```
PROC MAIN.ENTRY (CHAN OF SP fs, ts,
                 []INT free.memory,
                 []INT stack.memory)
```

Parameters are described in the following list.

fs Channel going from the host file server to the program ('from server').

ts Channel going from the program to the host file server ('to server').

free.memory Used by the program for its workspace. If the size of the **stack.memory** vector is zero then the **free.memory** vector is used for the program's run time stack as well as its static and heap data area, otherwise the vector is only used by the program for its static and heap data.

This vector represents the amount of free memory left after the program has been loaded. The size of this vector is determined from the environment variable **IBOARDSIZE** which specifies the amount of memory available on the transputer board (in bytes). The value of **IBOARDSIZE** is read at run time by the bootstrap loader before the program is started (see section 11.3.2).

stack.memory Used by the program for its run time stack if the size of the vector is non-zero.

The size of this vector is determined when the linked program is made bootable using **iboot**, by the parameter supplied with the '**S**' option.

File name	Transputer	Error mode
mainent.c4h	T414	HALT
mainent.c4s	T414	STOP
mainent.c4u	T414	UNDEFINED
mainent.c4x	T414	UNIVERSAL
mainent.c8h	T800	HALT
mainent.c8s	T800	STOP
mainent.c8u	T800	UNDEFINED
mainent.c8x	T800	UNIVERSAL

Table 9.1 Type 1 occam interface code files

9.3.2 Building a type 1 process

The type 1 occam interface code is supplied in the files **mainent.c4*x*** (for T414 transputers), and **mainent.c8*x*** (for T800 transputers). The full names of these files, along with the transputer types and error modes that they support, can be found in table 9.1.

For example, consider a C program that consists of the following object and library files:

```
main.bin
funcs.bin
crtl.lib
```

The program is to run on a T414 transputer in UNDEFINED mode. The files that make up the program are listed in the linker indirect file **prog.l4u**.

The program can be linked using one of the following commands:

```
ilink mainent.c4u /f prog.l4u /o cprog.c4u

ilink mainent.c4u -f prog.l4u -o cprog.c4u
```

When using the linker, the occam interface code file (**mainent.c4u** in the above example) should always be first file in the list, and an output file should be specified using the linker 'o' option. If you do not specify an output file, the linker uses the first filename in the list and adds a *cxx* extension. In the above example the default output file would be **mainent.c4u**, thereby overwriting the occam interface code.

When the program has been linked, you can use the bootstrap tool **iboot** to produce a bootable program, using the 'S' option to specify the amount of run time stack required. For the above example, using a stack size of 512 words, the command line would be one of the following:

```
iboot cprog.c4u /s 512

iboot cprog.c4u -s 512
```

9.4 Type 2 interface definition

The type 2 interface is used when building a program that will communicate with other processes as well as with the host file server. These processes can be running on the same processor or on other processors in the network and communicate with the program through channels.

The type 2 interface is used with C, FORTRAN or Pascal programs that have been linked with the full version of the respective run time library.

9.4.1 Type 2 procedural interface

The type 2 occam interface is defined as follows:

```
PROC PROC.ENTRY (CHAN OF SP fs, ts,
                 VAL INT flag,
                 []INT ws1, ws2,
                 []INT in, out)
```

Parameters are described in the following list.

fs Channel going from the host file server to the program.

ts Channel going from the program to the host file server.

flag Indicates the requirement for one or two work spaces. If the value of **flag** is set to zero then the program will run with two work space areas, one for static and heap data, the other for the run time stack. If the value of **flag** is set to one then the program will run with a single combined work space.

ws1 Used by the program for its workspace. If **flag** is zero then it is used only for the run time stack; if **flag** is one (1) then it is used as the program's combined workspace.

ws2 Used by the program as its static/heap workspace when **flag** is set to zero. Otherwise unused.

in A vector of pointers to occam channels going to the process.

out A vector of pointers to occam channels going from the process.

Note: The first two elements in the channel pointer vectors **in** and **out** are reserved for use by the C, FORTRAN or Pascal program's run time system and cannot be used by the program. See section 9.2.2 for more details on how these channel pointers are used.

9.4.2 Building a type 2 process

To build a type 2 equivalent occam process for a network program, link the compiled units together with type 2 interface code using **ilink**.

The type 2 occam interface code is supplied in the files **procent.c4***x* (for T414 transputers), and **procent.c8***x* (for T800 transputers). The full names of these files, along with the transputer types and error modes that they support, can be found in table 9.2.

To link the program use the linker '*newname*=' option to name the program. To call the program subsequently from the occam source, you must use this name. The names of all equivalent occam processes in an occam program must be unique.

For example, consider a C program made up of the following object and library files:

```
main.bin
funcs.bin
crtl.lib
```

File name	Transputer	Error mode
procent.c4h	T414	HALT
procent.c4s	T414	STOP
procent.c4u	T414	UNDEFINED
procent.c4x	T414	UNIVERSAL
procent.c8h	T800	HALT
procent.c8s	T800	STOP
procent.c8u	T800	UNDEFINED
procent.c8x	T800	UNIVERSAL

Table 9.2 Type 2 occam interface code files

This program is for a T800 transputer, running in UNDEFINED error mode, and the files that make up the program are listed in the linker indirect file **prog.l8u**.

The program can be linked using one of the following commands:

```
ilink cproc1=procent.c8u /f prog.l8u /o cproc1.c8u

ilink cproc1=procent.c8u -f prog.l8u -o cproc1.c8u
```

The '**cproc1=**' part of the command line gives the name **cproc1** to the equivalent occam process produced. This is the name that the program will be called from the occam source and must be a legal occam name. The occam interface code file (**procent.c8u**) should always be the first file in the list.

The linker '**o**' option is used to specify the output file name for the equivalent occam process. If you do not specify an output file name, the linker uses the first file name in the list, and appends a **.c***xx* suffix. If no output file had been specified in the above example, then the linker would have written to **procent.c8u**.

Having built an equivalent occam process you can then call it from occam source using the **#IMPORT** directive. It is recommended that processes written in C, FORTRAN and Pascal are wrapped in a small occam procedure with a *clean* channel interface (rather than arrays of channel addresses directly). The following section contains two examples to illustrate how this can be achieved.

9.4.3 Example type 2 wrappings

This section contains examples of how to call a type 2 equivalent occam process from occam source, and how to set up the parameters required.

Example 1: simple C call

The following example is of the occam procedure '`call.prog1`', within which a C program is called. The C program has already been made into an equivalent occam process in the object file '`cproc1.c8u`'. The call name is `cproc1`.

The source of this procedure is supplied with the toolset examples in the file `ctype2a.occ`.

```
PROC call.prog1 (CHAN OF SP fs, ts)

  #INCLUDE "hostio.inc"
  #IMPORT "cproc1.c8u"  -- C program cproc1()

  VAL flag IS 1 :       -- combined heap and stack

  [100000]INT ws1 :     -- stack and heap for program
  [1]INT ws2 :          -- dummy workspace for program
  [2]INT in, out :      -- channel pointers

  -- call program
  cproc1(fs, ts, flag, ws1, ws2, in, out)

:
```

After the occam wrapping has been compiled it can then be linked with the C process in the following way to generate the output file `ctype2a.c8u`:

```
ilink ctype2a.t8u cproc1.c8u
```

The resulting linked code could then be called, like any other separately compiled unit, by the `#USE` directive.

Example 2: C call setting up user channels

The following example is of the occam procedure '`call.prog2`', within which a C program is called. The C program has already been made into an equivalent occam process in the file '`cproc2.c8u`'. The call name is `cproc2`.

The source of this procedure is supplied with the toolset examples in the file `ctype2b.occ`.

```
#INCLUDE "hostio.inc"
PROC call.prog2 (CHAN OF SP fs, ts,
                 CHAN OF COMM to.process,
                 CHAN OF COMM from.process)

  #IMPORT "cproc2.c8u"  -- C program cproc2()

  VAL flag IS 0 :       -- separate heap and stack

  [1000]INT ws1 :       -- stack for program
  [40000]INT ws2 :      -- heap for program
  [3]INT in, out :      -- pointers to inputs/outputs

  SEQ

    -- set up user output channel
    LOAD.OUTPUT.CHANNEL(out[2], from.process)

    -- set up user input channel
    LOAD.INPUT.CHANNEL(in[2], to.process)

    -- call program
    cproc2(fs, ts, flag, ws1, ws2, in, out)

:
```

Two channels are declared of type COMM, the first being an input channel to the process, the second an output channel from the process. The declaration of protocol type COMM is assumed.

The first statement sets up a pointer to the output channel, using LOAD.OUTPUT.CHANNEL. The second statement sets up a pointer to the input channel, using LOAD.INPUT.CHANNEL.

9.5 Type 3 interface definition

The type 3 interfaces, like the type 2 interface, are used to run programs which communicate with other processes on the same processor or in a network of processes, but which do not require access to host services. Processes built using the type 3 interfaces can communicate with other processes through channels in the same way as for type 2 processes.

Four type 3 interfaces are provided, one each for C, FORTRAN and Pascal programs, and one that can be used with all three of the languages.

The three specific interfaces are used with programs linked with reduced versions of their respective run time libraries. The fourth or *stub* version of the interface can only be used with the full run time libraries.

9.5.1 Type 3 procedural interfaces

The four interfaces for type 3 equivalent occam processes are defined below.

For C programs:

```
PROC PROC.ENTRY.RC (VAL INT flag,
                    []INT ws1, ws2,
                    []INT in, out)
```

For FORTRAN programs:

```
PROC PROC.ENTRY.RF (VAL INT flag,
                    []INT ws1, ws2,
                    []INT in, out)
```

For Pascal programs:

```
PROC PROC.ENTRY.RP (VAL INT flag,
                    []INT ws1, ws2,
                    []INT in, out)
```

For C, FORTRAN or Pascal programs linked in with their full run time libraries:

```
PROC PROC.ENTRY.STUB (VAL INT flag,
                      []INT ws1, ws2,
                      []INT in, out)
```

Parameters are described in the following list.

flag Indicates the requirement one or two work spaces. If the value of **flag** is set to zero then the program will run with two work space areas, one for static and heap data, the other for the run time stack. If the value of **flag** is set to one then the program will run with a single combined work space.

ws1 Used by the program for its workspace. If **flag** is zero then it is used only for the run time stack; if **flag** is one then it is used as the program's combined workspace.

ws2 Used by the program as its static/heap workspace when **flag** is set to zero. Otherwise it is unused.

in A vector of pointers to occam channels going to the process.

out A vector of pointers to occam channels coming from the process.

Note: The first two elements in the channel pointer vectors **in** and **out** are reserved for use by the C, FORTRAN or Pascal program's run time system and cannot be used by the program. See section 9.2.2 for more details on how these pointers are used.

9.5.2 Building a type 3 process

To build a type 3 equivalent occam process for a network program, link the compiled units together with the appropriate interface code using the toolset linker, **ilink**.

The type 3 occam interface code is supplied in the files **procent/.c4***x* (for T414 transputers) and **procent/.c8***x* (for T800 transputers), where **l** is the language version (**c** for C, **f** for FORTRAN, **p** for Pascal, and **s** for universal). The full names of these files, along with the transputer types and error modes that they support, can be found in tables 9.3, 9.4, 9.5 and 9.6.

To link the program use the linker '*newname*=' option to name the program. To call the program subsequently from the occam source, you must use this name. The names of all equivalent occam processes in an occam program must be unique.

For example, consider a C program made up of the following object and library files:

```
main.bin
funcs.bin
sacrtl.lib
```

These files are listed in the linker indirect file **prog.l8u**.

C specific versions		
File name	**Transputer**	**Error mode**
procentc.t4h	T414	HALT
procentc.t4s	T414	STOP
procentc.t4u	T414	UNDEFINED
procentc.t4x	T414	UNIVERSAL
procentc.t8h	T800	HALT
procentc.t8s	T800	STOP
procentc.t8u	T800	UNDEFINED
procentc.t8x	T800	UNIVERSAL

Table 9.3 Type 3 C occam interface code files

FORTRAN specific versions		
File name	**Transputer**	**Error mode**
procentf.t4h	T414	HALT
procentf.t4s	T414	STOP
procentf.t4u	T414	UNDEFINED
procentf.t4x	T414	UNIVERSAL
procentf.t8h	T800	HALT
procentf.t8s	T800	STOP
procentf.t8u	T800	UNDEFINED
procentf.t8x	T800	UNIVERSAL

Table 9.4 Type 3 FORTRAN occam interface code files

Pascal specific versions		
File name	**Transputer**	**Error mode**
procentp.t4h	T414	HALT
procentp.t4s	T414	STOP
procentp.t4u	T414	UNDEFINED
procentp.t4x	T414	UNIVERSAL
procentp.t8h	T800	HALT
procentp.t8s	T800	STOP
procentp.t8u	T800	UNDEFINED
procentp.t8x	T800	UNIVERSAL

Table 9.5 Type 3 Pascal occam interface code files

C, FORTRAN and Pascal versions		
File name	**Transputer**	**Error mode**
procents.c4h	T414	HALT
procents.c4s	T414	STOP
procents.c4u	T414	UNDEFINED
procents.c4x	T414	UNIVERSAL
procents.c8h	T800	HALT
procents.c8s	T800	STOP
procents.c8u	T800	UNDEFINED
procents.c8x	T800	UNIVERSAL

Table 9.6 Type 3 C, FORTRAN, and Pascal occam interface code files

The program can be linked using one of the following commands:

```
ilink cproc1=procentc.t8u /f prog.l8u /o cproc1.c8u

ilink cproc1=procentc.t8u -f prog.l8u -o cproc1.c8u
```

The '**cproc1=**' part of the command line gives the name **cproc1** to the equivalent occam process produced. This is the name that the program will be called from the occam source and must be a legal occam name. You *must* always specify a name in this way, and each name must be unique. The occam interface code file **procentc.t8u** should always be the first file in the list.

The linker 'O' option is used to specify the output file name for the equivalent occam process. If you do not specify an output file name, the linker uses the first file name in the list, and appends a *.cxx* suffix. If no output file had been specified in the above example, then the linker would have written to **procentc.c8u**.

Having built an equivalent occam process you can then call it from occam source using the **#IMPORT** directive. It is recommended that processes written in C, FORTRAN and Pascal are wrapped in a small occam procedure with a *clean* channel interface (rather than arrays of channel addresses directly). The following section gives an example of how this can be achieved.

9.5.3 Example type 3 wrapping

This section shows how to call an equivalent occam process from occam source, and how to set up the parameters required.

The following example is of the occam procedure '**call.prog**', within which a C program is called. The C program has already been made into an equivalent occam process, and given the name '**cproc1.c8u**'.

The source of this procedure is supplied with the toolset examples in the file **ctype3.occ**. The procedure is listed below.

```
PROC call.prog (CHAN OF COMM to.process,
                CHAN OF COMM from.process)

  #IMPORT "cproc1.c8u"  -- C program cproc1()

  VAL flag IS 0 :       -- separate heap and stack

  [1000]INT ws1 :       -- stack for program
  [40000]INT ws2 :      -- heap for program
  [3]INT in, out :      -- pointers to inputs/outputs

  SEQ

    -- set up user output channel
    LOAD.OUTPUT.CHANNEL(out[2], from.process)

    -- set up user input channel
    LOAD.INPUT.CHANNEL(in[2], to.process)

    -- call program
    cproc1(flag, ws1, ws2, in, out)

:
```

Two channels are declared of type COMM, the first being an input channel to the process, the second an output channel from the process. The declaration of protocol type COMM is assumed.

The first statement sets up a pointer to the output channel, using the routine **LOAD.OUTPUT.CHANNEL**. The second statement sets up a pointer to the input channel, using the routine **LOAD.INPUT.CHANNEL**.

See section 4.4.4 for information on mixing error modes.

9.6 Channel communication

Communication between an equivalent occam process and any other process is via occam channels. This section contains some guidelines for implementing channel communication in non-occam languages, and introduces some of the features used in the INMOS implementations of C, FORTRAN and Pascal for the transputer.

9.6.1 Communication libraries

Routines to assist with implementing occam-like protocols in C, FORTRAN and Pascal are provided in channel i/o packages that are supplied as part of the run time libraries for INMOS-compatible compilers for the languages. The packages provide routines equivalent to the occam message passing primitives ! and ?. Routines that implement other protocols may also be available and for details of these consult the compiler documentation.

The following list contains functional definitions of the routines provided in the non-occam run time libraries for implementing primitive occam protocols. The routines are defined using a functional description in a simple meta-language form. Mnemonic names are used to express the functions of the routines and their parameters and each definition is followed by its occam equivalent.

The list of functional definitions is followed by details of the routines provided in the run time libraries for INMOS-approved C, FORTRAN and Pascal.

outbyte (byteval, channelid)

Send a byte value down a specified channel. occam equivalent:

```
channel ! BYTE byteval
```

outword (wordval, channelid)

Send a 32-bit integer value down a specified channel. occam equivalent:

```
channel ! INT32 wordval
```

outmess (channelid, bytevec, veclen)

Send a byte array down a specified channel. occam equivalent:

```
channel ! [bytevec FROM 0 FOR veclen]
```

inmess (channelid, bytevec, veclen)

Receive a byte array from a specified channel. occam equivalent:

```
channel ? [bytevec FROM 0 FOR veclen]
```

These four routines correspond directly to the channel i/o instructions supported on the transputer and to the channel input and output statements in occam. Other protocols can be implemented by using these routines in a particular way. For example, to provide channel input routines to correspond with *outbyte* and

outword, the function *inmess* can be used in the following way:

Data type	Function call
BYTE	*inmess (channelid, byteval, 1)*
INT32	*inmess (channelid, wordval, 4)*

9.6.2 C channel communication

In C, occam channels are identified by pointers to variables or array elements and are passed through arguments to the *main* function. It may be convenient to define a channel type consisting of an array of pointers.

The primitive channel communication functions provided in the C run time library are given below. Other routines for channel communication may be provided with the compiler you are using. For details of these routines and for further information about channel communication in C see the user documentation that is provided for the compiler.

```
_outbyte(byte, chanpointer)

_outword(word, chanpointer)

_outmess(chanpointer, buffer, nbytes)

_inmess(chanpointer, buffer, nbytes)
```

9.6.3 FORTRAN channel communication

FORTRAN accesses channels by index values which correspond to those of the occam vectors `in` and `out` in the equivalent occam process definition.

The primitive channel communication functions provided for FORTRAN are given below. Other routines for channel communication may be provided with the compiler you are using. For details of these routines and for further information about channel communication in FORTRAN see the compiler documentation.

```
SUBROUTINE CHANOUTBYTE (VALUE, ICHANNEL)
```

This is functionally equivalent to:

```
CHANOUTMESSAGE(ICHANNEL, BUFFER, 1)
```

```
SUBROUTINE CHANOUTWORD (VALUE, ICHANNEL)
```

This is functionally equivalent to:

```
    CHANOUTMESSAGE(ICHANNEL, BUFFER, 4)

SUBROUTINE CHANOUTMESSAGE (ICHANNEL, BUFFER, NBYTES)

SUBROUTINE CHANINMESSAGE (ICHANNEL, BUFFER, NBYTES)
```

9.6.4 Pascal channel communication

Pascal programs access channels by index values that correspond to those of the occam vectors **in** and **out** in the equivalent occam process definition.

You may also need to include the run time library at the head of the program and use special compiler options. For details of any special procedures you should use consult the compiler documentation.

The channel communication functions provided in the Pascal run time library are given below.

```
PROCEDURE outbyte (byte: CHAR; channel: INTEGER);

PROCEDURE outword (word, channel: INTEGER);

PROCEDURE outmess (channel: INTEGER;
                   VAR bufp: UNIV CHAR;
                   nbytes: INTEGER);

PROCEDURE inmess (channel: INTEGER;
                  VAR bufp: UNIV CHAR;
                  nbytes: INTEGER);
```

9.6.5 Implementing other occam protocols

Wherever possible you should use standard occam protocols with channels implemented in other languages. These protocols are not checked by normal language compilers and it is the programmer's responsibility to ensure that the protocols are adhered to.

You are recommended to write your own routines for protocol communication, using the low level channel routines provided by the run time libraries, rather than use the low level routines directly.

9.6.6 Guidelines and rules

When implementing channel protocols in other languages the occam rules of channel communication must be strictly adhered to because other languages do not provide a check on the way that channels are used. In particular, the length of a message transmitted from one process must match the length expected by the receiving process. If not, unpredictable results may occur.

Briefly, the rules for implementing occam protocols are as follows:

- All inputs must have a matched output and vice versa.
- Channels provide unbuffered, unidirectional point-to-point communication between two concurrent processes. The same channel must not be used for input and output.

The C, FORTRAN and Pascal primitive channel communication routines supplied in the run time libraries behave in the same way as occam input (?) and output (!), that is, communication does not proceed until the corresponding input or output is performed in the process at the other end of the channel.

The following sections show how to implement the three types of occam protocol, using the generalised routines that are described in section 9.6.1. For details of standard occam protocols, see the '*occam 2 Reference Manual*'.

Simple protocols

Simple protocols for communicating standard occam data types can be implemented directly using the primitive channel communication routines that are provided in the run time libraries. Table 9.7 and table 9.8 show how these routines can be called to implement standard occam protocols.

Arrays are input and output using routines *inmess* and *outmess*, respectively. Remember to adjust the length of *word* arrays by multiplying by the number of bytes per word.

Counted arrays should be preceded immediately by the length, using the appropriate output routine, as shown in table 9.7.

Sequential protocols

Sequential protocols are made up of a sequence of simple protocols. To implement a sequential protocol use a series of simple protocols in sequence. It is particularly important that the correct order is maintained, and you are strongly recommended to write routines for each type of sequential protocol that you wish

Data type	Routine
BYTE	*outbyte(byteval, channelid)*
BOOL	*outbyte(byteval, channelid)*
INT	*outword(wordval, channelid)* *outmess(channelid, bytevec, 4)*
INT16	*outmess(channelid, bytevec, 2)*
INT32	*outword(wordval, channelid)* *outmess(channelid, bytevec, 4)*
INT64	*outmess(channelid, bytevec, 8)*
REAL32	*outword(wordval, channelid)* *outmess(channelid, bytevec, 4)*
REAL64	*outmess(channelid, bytevec, 8)*

Table 9.7 Outputting simple protocols

Data type	Routine
BYTE	*inmess(channelid, bytevec, 1)*
BOOL	*inmess(channelid, bytevec, 1)*
INT	*inmess(channelid, bytevec, 4)*
INT16	*inmess(channelid, bytevec, 2)*
INT32	*inmess(channelid, bytevec, 4)*
INT64	*inmess(channelid, bytevec, 8)*
REAL32	*inmess(channelid, bytevec, 4)*
REAL64	*inmess(channelid, bytevec, 8)*

Table 9.8 Inputting simple protocols

to implement.

Variant protocols

A variant protocol specifies a number of possible formats for communication on a single channel. Each format has an identifying tag, which may be followed by

a simple or sequential protocol. For example:

```
PROTOCOL LINES
  CASE
    line; BYTE::[]BYTE
    error; INT16
    terminate
:
```

This defines a protocol called **LINES** which has three tags: **line** which is always followed by a byte counted array; **error** which is always followed by an **INT16**; and **terminate** which is not followed by any further data.

Variant protocol tags are byte values and are always numbered from zero. Thus in the above example the tag **line** would have the value 0, **error** would have the value 1, and **terminate** would have the value 2.

Each communication format in a variant protocol may be considered as a sequential protocol which always has a byte value as the first item sent. The value of this byte can be tested and the appropriate action taken.

For example, the following sequence would implement the **line** format communication from the **LINES** protocol:

outbyte(0, channelid)
outbyte(length, channelid)
outmess(channelid, line, length)

The following sequence would implement the **error** format communication from the **LINES** protocol:

outbyte(1, channelid)
outmess(channelid, number, 2)

The following sequence would implement the **terminate** format communication from the **LINES** protocol:

outbyte(2, channelid)

9.7 Calling occam from other languages

occam procedures can be called from C, FORTRAN and Pascal if the following rules are observed.

1 The occam procedure to be called should be a separately compiled unit.

2 occam FUNCTIONs cannot be called.

3 A parameter of type INT should be declared as the first formal parameter for the occam procedure. The parameter is not used in the occam procedure, but is used by C, FORTRAN and Pascal as the static link.

4 Formal parameters for the occam procedure must be scalars or arrays and must *not* be VAL parameters. Array bounds must be declared within the occam procedure. Dynamically sized arrays can be passed into occam by oversizing the formal array parameter in occam, specifying an additional formal parameter to define the actual size, and setting up an abbreviation in the occam procedure to define the bounds, thereby enabling subscript checking.

5 Actual parameters should be passed from the calling program as pointer values. In C this can be a pointer to an integer or the address of an integer, and in Pascal a pointer variable. In FORTRAN all parameters are implicit pointers.

6 Interaction with the calling program must be via channel parameters. Channel parameters are passed from C programs as pointer values. Standard FORTRAN and Pascal have no way of passing channel equivalents, and so interaction of occam with a calling FORTRAN or Pascal program is not possible without special extensions. Extensions to create channel pointers may be available with some versions of parallel FORTRAN compilers. For further details consult the compiler documentation.

7 The occam procedure must not use its vector space. If arrays are used they should be explicitly placed within the workspace.

8 It is the programmer's responsibility to ensure that there is enough workspace on the stack of the calling program.

As occam is a static language without global variables (unlike C, FORTRAN and Pascal) it is possible that any common occam routines can be shared between the C, FORTRAN and Pascal programs that use them and that also run on the same processor.

This can be achieved by first linking the C, FORTRAN or Pascal programs without the occam routines that they have in common using the linker '-U' option (allowing unresolved references). The linked C, FORTRAN or Pascal programs, the common occam routines that are to be shared and the occam wrapping that calls the C, FORTRAN or Pascal programs, can then be linked together to produce the entire program.

9.7.1 Examples

The following examples show how C, FORTRAN and Pascal programs can call a occam routine which has integer and byte variables as part of its formal parameter specification.

The formal parameter specification for such an occam routine is shown below:

```
PROC occam1 (INT dummy, INT word, BYTE byte)
```

Calling sequences for C, FORTRAN, and Pascal are illustrated below.

C:

```
int word;
char byte;

extern void occam1();

occam1(&word, &byte);
```

FORTRAN:

```
INTEGER WORD
CHARACTER BYTE

EXTERNAL OCCAM1

CALL OCCAM1(WORD, BYTE)
```

Pascal:

```
VAR
    word : INTEGER;
    byte : CHAR;

IMPORT PROCEDURE occam1 ALIAS 'occam1'
                        (VAR word : INTEGER;
                         VAR byte : CHAR);

BEGIN
    occam1(word, byte);
END.
```

The following examples show how C, FORTRAN and Pascal programs can call an occam routine which has integer and byte vector variables as part of its formal parameter specification.

The formal parameter specification for such an occam routine is as follows:

```
PROC occam2 (INT dummy,
             [10]INT words,
             [20]BYTE bytes)
```

Calling sequences for C, FORTRAN, and Pascal are illustrated below.

C:

```
int words[10];
char bytes[20];

extern void occam2();

occam2(words, bytes);
```

FORTRAN:

```
INTEGER WORDS
CHARACTER BYTES

DIMENSION WORDS(10), BYTES(20)

EXTERNAL OCCAM2

CALL OCCAM2(WORDS, BYTES)
```

Pascal:

```
TYPE
    wordvector = ARRAY [0..9] OF INTEGER;
    bytevector = PACKED ARRAY [0..19] OF CHAR;

VAR
    words : wordvector;
    bytes : bytevector;

IMPORT PROCEDURE occam2 ALIAS 'occam2'
                        (VAR words : wordvector;
                         VAR bytes : bytevector);

BEGIN
    occam2(words, bytes);
END.
```

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.

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.

Setting the transputer 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 transputer's address space.

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 chapter 11, 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 4.4.6.

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 **n** to be allocated address **#80000004** on a 32-bit transputer:

```
PLACE n 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 (**[]INT** subscript 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.

The following two code fragments illustrates 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:

```
[4]CHAN OF ANY out.links, in.links :

PLACE   out.links AT 0:
PLACE   in.links  AT 4:
```

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.

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 boundary, the object must be an element of an array which is placed on a word address below the required address. For example, to access a **BYTE** port called **io.register** located at physical address **#40000001** on a T414 the following 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.

10.2 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.

10.2.1 Using the code insertion mechanism

An occam 2 code insertion is introduced by the construct **GUY**. The context of the **GUY** construct is determined, as with all occam constructs, by the text indentation. The transputer instructions which follow the **GUY** 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 appendix B.

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:

```
GUY
  LDC 0
  LDL a
  LDL b
  LSUM
  SUM
  STL c
```

which would avoid the storing and reloading of **carry** and **temp**.

Values in the range **MOSTNEG INT - 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. This means that a single line of code such as `LDL a` might be translated by the compiler into a sequence of transputer instructions.

A more complex example, which sets 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
    GUY
      LDL    a     -- push value of free variable
                   --       onto stack
      LDC    512   -- push 512 onto stack
      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 `GUY` construct, in which case, the work space locations are accessed like any other occam symbol.

```
INT   a :
SEQ
  INT   b, c :
  GUY
    LDL    a      -- push value in a onto stack
    STL    b      -- pop value from stack into b
    ...  more code
```

10.2.2 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. Then when the label is used in an instruction, precede the name with a full stop. For example:

```
GUY
  ...  some instructions
  :FRED
  ...  some more instructions
  CJ .FRED
```

The same label name may not be defined more than once within an occam procedure.

10.3 Dynamic code loading

The toolset compiler permits the dynamic loading and execution of code using the procedures described in this section.

The dynamic code loading 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.

The procedures are outlined in the table below, and described in the following sections, with examples.

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.vec
LOAD.OUTPUT.CHANNEL	INT here, CHAN OF ANY out
LOAD.OUTPUT.CHANNEL.VECTOR	INT here, []CHAN OF ANY out.vec
LOAD.BYTE.VECTOR	INT here, []BYTE b.vec

The bootstrap tool **iboot** described in chapter 11, can produce code in a format suitable for dynamic loading. The file format is described in appendix E.

10.3.1 Calling code

The occam 2 compiler recognises 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 loaded 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 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 at the end of the parameter list in **workspace**. This pointer is treated like an extra parameter and so **no.of.parameters** must be increased by one.

The value of the integer **no.of.parameters** should be increased by one if separate vector space is used.

The workspace passed to **KERNEL.RUN** must be at least:

```
[ws.requirement + no.of.parameters + 3]INT
```

or if the program does not require separate vector space:

```
[ws.requirement + no.of.parameters + 2]INT
```

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 immediately above the saved **Iptr** word, and the last parameter should be in the word immediately below the saved **Wptr** word.

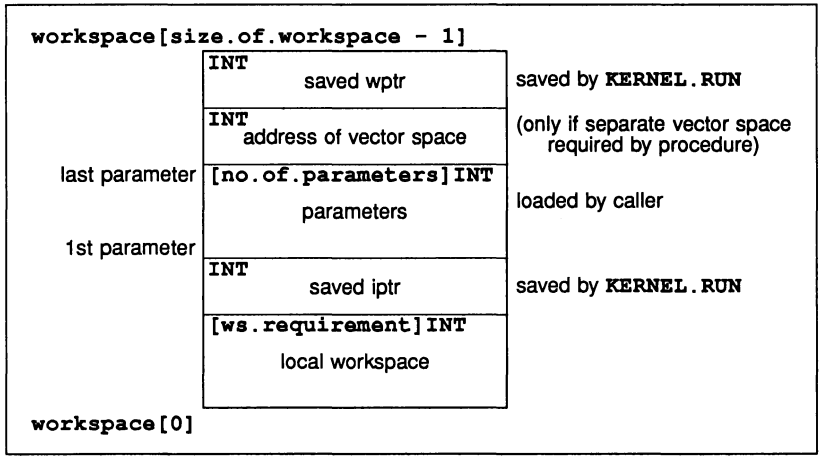


Figure 10.1 Workspace buffer

10.3.2 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.vec)
```

The variable **here** is assigned the address of the input channel vector **in.vec**.

```
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.vec)
```

The variable **here** is assigned the address of the output channel vector **out.vec**.

```
LOAD.BYTE.VECTOR (INT here, []BYTE b.vec)
```

The variable **here** is assigned the address of the byte vector **b.vec**.

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.

10.3.3 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 input to and output from 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 bootstrap tool **iboot** (chapter 11), 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 6:  -- No. of parameter slots
  -- Need 1 slot per parameter + 1 for the size of the
  -- array parameter

  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 process. 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.

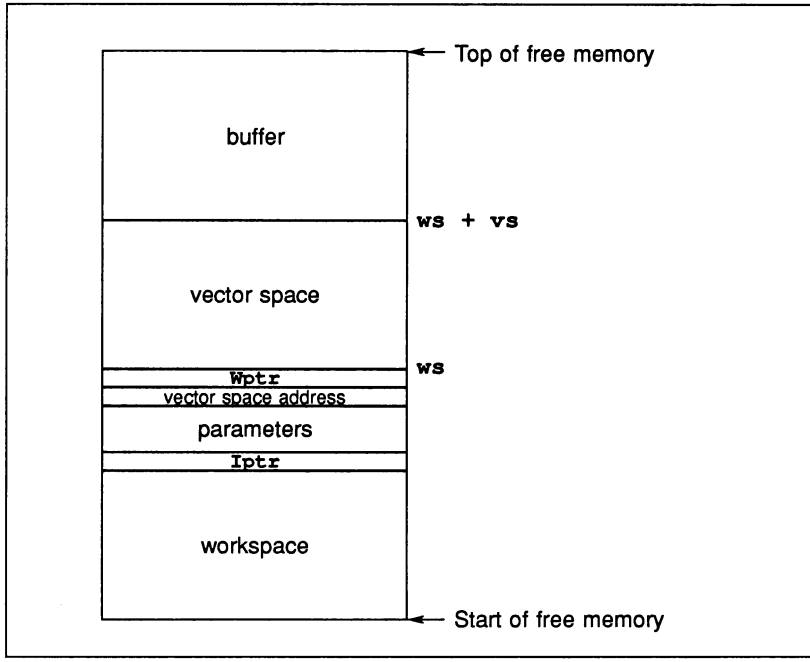


Figure 10.2 Partitioning of free memory

10.4 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 section 24.9.

10.4.1 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.

10.4.2 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.

10.4.3 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.

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)
```

10.4.4 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.

10.4.5 Example: a development system

For our example consider the development system described in section 10.4.1, 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.

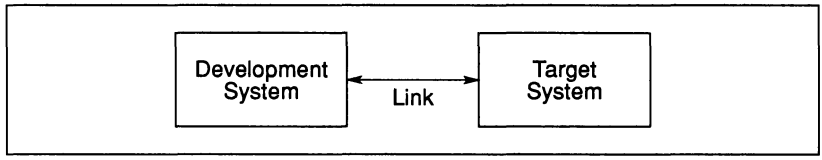


Figure 10.3 Development system

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.

10.5 Setting the error flag

The transputer error flag can be set using the predefined procedure **CAUSEERROR()**. This procedure is recognised automatically by the compiler and does not need to be referenced by the **#USE** directive.

CAUSEERROR always halts the program, whatever the mode of the compilation. This is distinct from the occam primitive process **STOP**, which only halts the program if the compilation is in HALT mode.

If the program was loaded using the **iserver** '**SE**' option, the server terminates when the error flag is set.

Reference manual

72 TDS 184 00

April 1989

11 iboot — bootstrap tool

This chapter describes the bootstrap tool that produces executable code for single transputers. It shows how to invoke the tool, describes the function and format of the bootstrap loader program and explains how it allocates memory. The chapter also describes how to create new bootstrap loader programs and ends with a list of error messages.

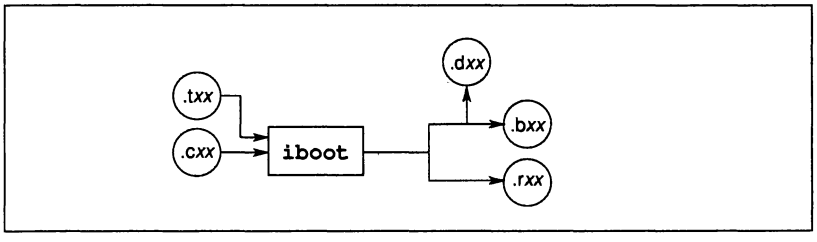
11.1 Introduction

The bootstrap tool generates an object file that can be loaded onto a single transputer and run. This file is known as a **bootable** file. It can also generate a file suitable for loading by a program at run time (dynamic loading). The input file can be a compiled (`.txx`) or linked (`.cxx`) object file.

The bootstrap tool can be considered a simple configuration program for single transputers. It converts a linked program image into a bootable code file that will run on a single transputer, by adding bootstrap code to the program.

The bootstrap tool can also be used to create new bootstrap loader programs for special uses.

The operation of the bootstrap tool in terms of file extensions is illustrated below.



11.1.1 Programs that can be made bootable

Programs to be made into bootable code must have a procedural interface as defined in section 11.3.2, and the program must run on a single transputer.

The following files cannot be made into bootable programs:

- Object files created by the librarian
- Object files containing unresolved references to other compilation units
- Object files that contain more than one entry point.

11.1.2 Transputer targets

The bootstrap tool produces code for the IMS T212, T222, M212, T414, T425, and T800 transputers. If the target transputer is of any other type an error is reported.

11.2 Running the bootstrap tool

The syntax of the bootstrap tool is as follows:

iboot *filename* {*options*}

where: *filename* is the name of the input file. The file extension must be given.

options is a list, in any order, of one or more of the options listed in Table 11.2.

11.2.1 Bootstrap code

The exact code added by the bootstrap tool depends on the transputer target and error mode of the input object file. The same loader is used for all three 32-bit processors (T414, T425, and T800), but different code is used for the IMS 212. The function of both loaders is identical.

If the input object file is compiled in HALT system mode then the halt-on-error flag is set in the bootstrap loader code; if the file is compiled in any other mode then the halt-on-error flag is not set. Option '**E**' can be used to disable the setting of the halt-on-error flag in HALT mode programs, and to enable the setting of the halt-on-error flag in other modes.

11.2.2 Producing code for dynamic loading

To produce code that can be loaded dynamically at run time, use the '**R**' option to prevent the addition of normal bootstrap code. The resulting file is non-bootable and contains all the information required for the call to the program, plus the required code (in binary format). The file is given the *.rxx* extension. The format of non-bootable files is described in appendix E.

Option	Description
C	Produces bootable code for input to the configurer programs for C, FORTRAN and Pascal code. For details of the specific configurer programs, see the documentation for the language compiler.
E	Disables the setting of the halt-on-error flag for programs compiled in HALT mode and enables it for other modes.
I	Displays brief progress information as the tool runs.
M	Disables production of a code map file.
P	Creates a new bootstrap loader program.
R	Disables the addition of bootstrap loader code to program code. Used to generate code for dynamic loading and for booting from ROM.
O *outputfile*	Specifies an output filename. If no filename is specified the input filename is used and the **.b***xx* extension is added.
B *filename*	Specifies an external bootstrap loader.
S *size*	Specifies the size of the run time stack for C, FORTRAN, and Pascal programs. **size** must be specified in words not bytes. The default is to use a combined stack.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. Spaces between the options and the case of letters in parameters are not significant. Options may be specified in any order.	

Table 11.1 Command line options

For more information about dynamic loading see section 10.3.

11.2.3 External loaders

External bootstrap loaders can be used in place of the default bootstrap code by specifying the 'B' option. The file produced is of the same format as a non-bootable file generated using the 'R' option, except that the external bootstrap code is added to the file.

Note: Programs that contain external loader code cannot be debugged with the toolset debugger **idebug** unless they use the same loading scheme as the default loader, described in section 11.3.2.

New bootstrap loaders can be created using the 'P' option. For more details see section 11.4.

11.3 Bootstrap loaders

The function of the bootstrap loader is to initialise the transputer, set up loader and program workspaces, and load the program. The loader consists of two parts: the *primary* loader, and the *secondary* loader.

The primary loader initialises the transputer and passes control to the secondary loader, which loads the program. The same primary loader is used on all 16-bit and 32-bit transputers.

The secondary loader sets up workspaces and code areas for the program, loads the code, and calls the program. The exact object code used varies between 32-bit and 16-bit transputers.

The interfaces between the primary loader and the secondary loader are described below.

11.3.1 Secondary loader interface

The secondary loader program must conform to the following procedure definition:

```
PROC procname  (CHAN OF ANY from.link,
                CHAN OF ANY to.link,
                VAL INT bytes.per.word,
                VAL INT word.mem.start,
                VAL INT word.free.mem.offset)
```

where: `from.link` and `to.link` are the input and output channels respectively of the transputer link down which the transputer was booted.

`bytes.per.word` is the number of bytes per word on which the process is to run (4 for the T414, T425 and T800 and 2 for the T212, T222 and M212).

`word.mem.start` is the word offset of `MemStart` from `MOSTNEG INT` (18 for the T414, T212 and T222, 28 for the T425 and T800).

`word.free.mem.offset` is the word offset from `MOSTNEG INT` of the start of *free* memory, that is, memory unused by the primary and secondary loader code and workspaces.

The third and fourth parameters are determined by the transputer type.

11.3.2 Program interface

The program must conform to one of the following procedure declarations:

```
PROC program (CHAN OF ANY from.link, to.link,
              []INT user.buffer)

PROC program (CHAN OF ANY 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 the stack buffer for non-occam code. The stack buffer is only used if space is reserved for a separate non-occam stack using the 'S' option. The stack is allocated at the base of memory. If the 'S' option is not specified when **iboot** is invoked a combined run-time stack and static and heap workspace is used (allocated from **user.buffer**).

The last parameter 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.

11.3.3 Memory allocation

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 Program code is placed as low as possible in memory, taking into account the scalar work space requirement of the program and the requirements of the separate C, FORTRAN and Pascal stack if the **iboot** 'S' option has been selected.

2 Program code is placed above the memory required for the program's scalar work space and the C, FORTRAN and Pascal stack if specified. The memory reserved by the loader for the program's code and scalar workspace will overlap the loader's own work and code space.

3 If the program uses a separate vector workspace the loader reserves a portion of the program's memory as vector workspace. From the size of this workspace, the size of the program code, and the size of its scalar 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 for the program to use.

4 If the program uses a separate stack for C, FORTRAN or Pascal code, the stack is placed at the base of memory, beginning at **MemStart**.

Figure 11.1 shows the memory map of the loaded code as created by the default bootstrap loader.

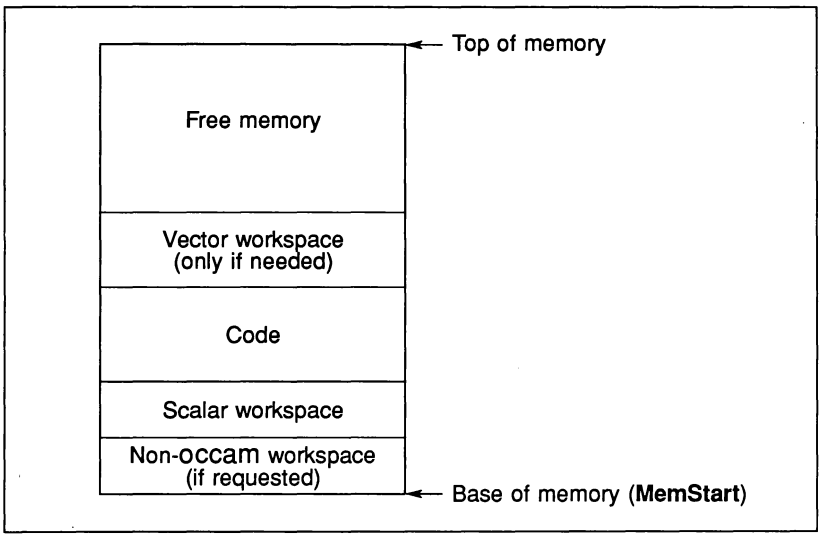


Figure 11.1 Memory map

11.4 External bootstrap loaders

External bootstrap loaders can be created for special purposes, for example when programs are to be run on non-standard transputer boards, or when pro-

grams require special code and workspace allocation. External loader programs are specified to the bootstrap tool using the 'B' option.

Programs that will be booted with external loaders must conform to the interface required by that loader program. A description of the interface used by the primary loader can be found in section 11.3.1.

Note: When external bootstrap loaders are used no check is made on the formal parameter specification of the main entry point.

11.4.1 Creating external loaders

External bootstrap loaders are created, like bootable programs, using the `iboot` tool. To create a new loader program, invoke `iboot` and specify the loader program file using the 'P' option. A new bootstrap loader is created from the file by adding the fixed primary loader code, and the result is written to an output file. This file can then be used in place of the default loader program by specifying the 'B' option.

When creating new loader programs, all the normal bootstrap options are available. For example, the 'E' option can be used to set or clear the transputer error flag (depending on the error mode of the program) when the program encounters an error.

11.5 Error messages

Other messages not in this list may be generated by corrupt files and by files not created by the toolset.

Bootstrap file name too long (*value*)

> Command line error. *value* is the number of characters read. The maximum length for bootstrap file names is 255 characters.

Cannot allocate stack space (*value*)

> The 'C' and 'S' options hav been used together. These two options are mutually exclusive. *value* is the size of the stack space required.

Cannot allocate vector space (*value*)

> The input object file requires a separate vector space and the 'C' option has been specified. *value* is the size of the vector space required.

Code not contiguous, *address*

The transputer code contained in the input object file is not contiguous, that is, the code is stored randomly in memory. This error is generated if unlinked C, FORTRAN or Pascal code is specified as input.

Illegal formal parameter specification

The list of parameters for the main entry point of the program does not match that required by the bootstrap loader.

Illegal link data tag, *tagtype*

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

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

Illegal option (*char***)**

An unknown option was specified. *char* is the illegal option character.

Illegal processor type, *type*

The input object file defined an illegal target processor type (TA, TB or TC). The bootstrap tool accepts the processor types T212, T414, T425 and T800.

Input file name too long (*value***)**

Command line error. *value* is the number of characters read. The maximum length for input file names is 255 characters.

Multiple bootstrap files specified

More than one bootstrap loader file was specified when using the '**B**' option.

Multiple definition, ENTRY

More than one **ENTRY** data tag was found in the input object file. This can occur if the file has more than one main entry point.

Multiple input files specified

More than one input file was specified.

Multiple output files specified

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

Multiple stack sizes specified

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

Output file name too long (*value*)

Command line error. *value* is the number of characters read. The maximum length for output file names is 255 characters.

No bootstrap file specified

No file was specified after the 'B' option.

No input file specified

No input file was specified.

No output file specified

No output file was specified after the 'O' option.

No stack size specified

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

Unable to close (*value*)

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

Unable to open (*value*)

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

Unable to read (*value*)

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

Unable to write (*value*)

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

12 icheck — occam 2 checker

This chapter describes the occam 2 checker tool **icheck** that enables you to check programs for correct syntax before submitting them for compilation. In addition to the checks performed by the compiler, **icheck** performs an exhaustive check on the usage of variables and channels according to the rules of occam 2.

12.1 Introduction

The occam 2 checker performs a syntax check of the full occam 2 product language, as described in the '*occam 2 Reference Manual*', but produces no object code.

The syntax checking performed by the checker is similar to that of the compiler, but it generates more data and displays more information about the errors. By taking advantage of the regular structure of occam programs **icheck** can recover from errors that cause the compiler to abort, and thereby perform a more comprehensive check.

The checker recognises and checks the compiler directives **#INCLUDE**, **#USE**, **#IMPORT**, **#SC**, **#OPTION**, and **#COMMENT**. Directives are described in more detail in chapter 23.

Libraries and separately compiled units that are used within a program must already be compiled before the checker is used.

12.2 Running the checker

The checker takes as input an occam source file and performs syntax, alias and usage checks on the contents. Errors in the source text are displayed along with the file in which the error occurred, the line number, and a description of the error.

For each compilation the target processor type and the compilation error mode should be specified so that the checker can select the correct libraries. The checker assumes a compilation for the T414 in HALT mode. For this compilation the transputer target and error mode options can be omitted.

The checker recognises all of the compiler options, but ignores any that are not relevant to syntax checking.

To invoke the checker use the following command line:

icheck *filename* {*options*}

where: *filename* is the name of the file containing the source code. If no extension is given, the **.occ** extension is assumed.

options is a list, in any order, of any of the compiler options given in Tables 12.1 and 12.2.

Option	Description
TA	Check for transputer class TA (T414, T425, T800).
TB	Check for transputer class TB (T414, T425).
TC	Check for transputer class TC (T425, T800).
T2	Check for a T212 processor.
T212	Same as **T2**.
T222	Same as **T2**.
M212	Same as **T2**.
T4	Check for T414 processor. This is the default processor type and may be omitted when compiling for a T414 processor.
T414	Same as **T4** (default).
T5	Check for a T425 processor.
T425	Same as **T5**.
T8	Check for a T800 processor.
T800	Same as **T8**.
H	Checks code in HALT mode. This is the default mode and may be omitted for HALT mode programs.
S	Checks code in STOP mode.
U	Checks code in UNDEFINED mode.
X	Checks code in UNIVERSAL mode.
A	Disables alias checking. The default is to perform alias checking. This option also disables usage checking, which is dependent on alias checking. Details of alias and usage checking rules are given in the '*occam 2 Reference Manual*' and section 12.3 in this chapter.
B	Displays messages in brief (single line) format.

Table 12.1 occam 2 checker options

Option	Description
C	Disables code generation. A compiler option that is ignored by the checker.
D	Disables the production of debugging data. A compiler option that is ignored by the checker.
E	Disables the loading of the extended arithmetic libraries. A compiler option that is ignored by the checker.
G	Enables the checker to recognise the restricted range of transputer instructions, via the GUY construct. See appendix B for the list of permitted instructions.
I	Directs the checker to display additional run time information. The information includes version data and information about directives as they are processed. The default is not to display this information.
L	Loads the checker onto a board and terminates.
N	Disables usage checking. The default is to perform usage checking. Details of usage checking rules are given in the '*occam 2 Reference Manual*'.
O *outputfile*	Specifies the output file. A compiler option that is ignored by the checker.
R *filename*	Redirects error messages to a file.
V	Disables separate vector space. A compiler option that is ignored by the checker.
W	Enables the checker to recognise the full range of transputer instructions, via the GUY construct. See appendix B for a list of transputer instructions.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. Spaces between the options and the case of letters in parameters are not significant. Options may be specified in any order.	

Table 12.2 occam 2 checker options (contd)

12.2.1 Checker messages

Syntax errors are reported in the standard toolset format. For details of the format see section 3.7.5.

12.3 Alias and usage checking

The checks on variable and channel usage and on abbreviations performed by the checker conform closely to the rules of occam 2. Details of the rules can be found in the '*occam 2 Reference Manual*'.

The alias and usage checks performed by **icheck** are superior to those of the compiler. If the checker accepts code that the compiler rejects you can disable alias and usage checking in the compiler in order to get the program compiled. However, you should be aware that this also disables some useful run-time usage checks which are inserted by the compiler.

12.3.1 Usage checking

The checker implements the checking non-array and array elements in the same way as the compiler but with fewer restrictions. For example, the checker handles replicators correctly. For details of usage checking in the compiler see section 23.7.

12.3.2 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.

12.4 Error messages

This section lists error messages that can occur when the checker is invoked. It does not include the syntax error messages that are displayed by the checker in normal operation.

Cannot open file "*filename*"

The file *filename* does not exist or could not be opened.

Cannot open "*filename*" - too many nested SCs

Separately compiled units are nested too deeply.

Object code compiled in incompatible error mode (*errormode*)

The compilation error mode is incompatible with the main program. *errormode* is the mode for which the code was compiled.

Object code compiled for incompatible target (*processortype*)

The target transputer type is incompatible with the main program component. *processortype* is the type for which the code was compiled.

Object code not compatible with current version (*filename*)

The file referenced by a **#USE** or **#SC** directive was in a format incompatible with the checker. Check the version in use or recreate the file in the correct format.

13 iconf — configurer

This chapter describes the configurer tool **iconf** that allocates occam processes to processors and assigns channels to links on transputer networks. It explains how to invoke the configurer tool and describes the configuration description and language. The chapter ends with a summary of the configuration description and a list of error messages.

13.1 Introduction

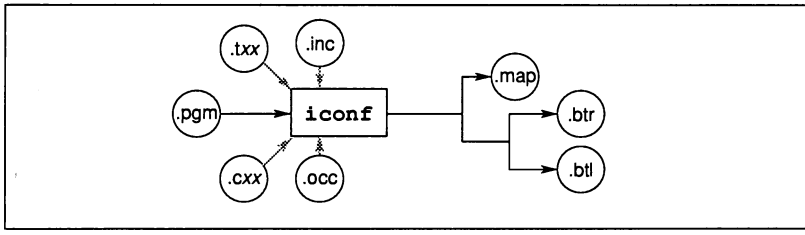
The configurer takes a configuration description and produces either an object code file ready for booting into a network of transputers, or a configuration map describing the allocation of code and placement of channels.

A configuration description describes how code is to be run on a network of transputers. It consists of occam-like configuration language statements that allocate code to processors and assign channels to transputer links.

Code to be run on separate processors can be declared as separately compiled units, or included as occam source. Separately compiled units that are to be run on the same transputer must be compiled for the same or a compatible transputer type, and in the same or a compatible error mode. Separately compiled code can be allocated to any number of transputers on the network.

The network can consist of any number of transputers, of any type, connected in any configuration. Each network contains a root processor, connected between the host and the transputer network, through which programs are loaded onto the network. There must be a route, via transputer links, from the root processor to all other processors on the network, to allow the program to be loaded.

The operation of the configurer tool in terms of toolset file extensions is illustrated below.



13.2 Running the configurer

The configurer takes as input a configuration description file consisting of configuration language statements and declarations, and produces object code allocated to specific processors.

To run the configurer use the following command line:

iconf *filename* {*options*}

where: *filename* is the file containing the configuration code. If no file extension is given **.pgm** is assumed.

options is a list of one or more options from table 13.1.

If no file name is specified brief help information is displayed.

13.2.1 Source compilation mode – options H S U

All source code in a configuration description is compiled in the same error mode.

The mode is determined by options on the command line. Options '**H**', '**S**', and '**U**' are used to ensure that all occam source in the configuration description is compiled in HALT, STOP or UNDEFINED mode respectively. The default is HALT mode.

Note: UNIVERSAL mode is not supported by the configurer.

See section 23.4 for more information on compilation error modes.

13.2.2 Generating a configuration map – option M

Option '**M**' directs the configurer to produce a configuration map file in place of a bootable code file. The file takes the same name as the configuration description file, but is given the **.map** extension.

The configuration map is a text file that contains information about the configuration description. It shows:

1 The memory layout for each processor.

2 The order in which the processors will be booted when the network is loaded.

3 The link connections between processors.

Option	Description
A	Disables alias checking when compiling occam source in the configuration description. Details of the alias checking performed by **iconf** are given in chapter 23.
G	Enables the configurer to compile code containing transputer instructions from the restricted set, inserted via the **GUY** construct. See appendix B for the list of permitted instructions.
H	Compiles occam source in HALT mode. HALT mode is also the default.
I	Displays progress information as the configurer runs.
L	Loads the configurer and terminates.
M	Generates a configuration map file showing code allocation on specific processors, in place of the normal output file. The file is given the **.map** extension.
N	Disables usage checking on occam source in the configuration description. Details of the usage checking performed by **iconf** are given in chapter 23.
O *filename*	Specifies an output filename. If you do not specify this option the input filename is used and the **.btl** or **.btr** extension is added.
R	Produces configuration code for loading onto ROM. If no output file is given and no extension specified the **.btr** extension is added. The default is to produce **.btl** code for boot from link boards.
S	Compiles occam source in STOP mode.
U	Compiles occam source in UNDEFINED mode.
V	Disables separate vector space.
W	Enables the configurer to compile code containing transputer instructions from the full set, inserted via the **GUY** construct. See the *Transputer instruction set: a compiler writer's guide* for a list of instructions.
X	Compiles occam source in UNIVERSAL mode.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. Spaces between options, and the case of letters in parameters, are not significant. Options may be specified in any order.	

Table 13.1 Configurer options

13.3 Configuration description

A configuration description consists of declarations and configuration language statements. It must declare the code to be run, allocate it to specific processors on the network, and assign channels to transputer links. The allocation of code and the mapping of channels to links enables the configurer to identify the code destined for a specific processor and to check the logical integrity of the whole network.

For an example of a configuration description see section 5.6.

13.3.1 Separately compiled code

Separately compiled code used in a configuration description must be referenced by the **#USE** directive. Where other compilation units or libraries are referenced within the separately compiled code, the code must also be linked using the **ilink** tool to produce a *.cxx* object file. Where no libraries or nested compilation units are used within the code, a *.txx* file can be used. If code is present that requires linking the configurer reports an error and the configuration is aborted.

Libraries may neither be directly referenced nor called in a configuration description. They should only be referred to in the compilation units in which they are used.

13.3.2 Source code

The configuration description can contain any occam source, but only within a **PROCESSOR** statement, as described in section 13.3.3. Outside a **PROCESSOR** statement only constants and protocol declarations are permitted. The configurer automatically compiles any source encountered in the configuration description before configuring the code on the network.

occam source files can also be included in a configuration description with the **#INCLUDE** directive. Outside a **PROCESSOR** statement **#INCLUDE** can only be used to refer to files containing constants and protocol declarations.

13.3.3 Configuration language

This section describes the configuration language statements that assign processes to processors, and map occam channels to transputer links. For more information about the configuration language statements, see the '*occam 2 Reference Manual*'.

Allocating code to processors

The allocation of code to processors is achieved using two configuration language statements:

```
PLACED PAR

PROCESSOR number transputertype
```

where: *number* identifies the processor on the network. *number* can be an integer or integer expression.

transputertype is the transputer type for which the code has been compiled. Valid transputer types are T212, T414, T425, T800, T2, T4, and T8. Transputer classes TA, TB and TC are not permitted. T212 should be used for IMS T222 and IMS M212 processors.

PROCESSOR statements contain the code for a single processor. The code can include any number of separately compiled units, or any legal occam source. Separately compiled code which has been compiled for a transputer class may be placed on any processor in the class.

Any legal occam source is permitted inside a **PROCESSOR** statement provided that no library routines are used and no libraries are referenced. The configurer compiles the occam source using the options specified on the command line, and adds it to the object code for that processor. If the source contains transputer instructions, option 'G' or 'W' should be specified.

Placing channels on links

Channels between processes are tied to specific transputer links using the **PLACE** statement. Absolute addresses of links are defined in the include file **linkaddr.inc**.

The syntax of the **PLACE** statement is as follows:

```
PLACE name AT address:
```

where: *name* is the name of a variable, channel, timer or array.

address is the address of a memory location.

Channels that support data exchange in both directions must be placed at an input link address on one processor and at an output link address on another processor, and must be declared outside the processor statements. Channels declared inside a **PROCESSOR** statement can still be placed at link addresses

but are not considered part of the wiring of the network. 'Dangling' links, that is, links on which only a single input or output channel has been placed produce a warning message but are not prohibited by the configurer.

13.4 Summary of configuration description

A configuration description has the form:

Configuration-level declarations
PLACED PAR
PROCESSOR *statements*
process code

Permitted configuration level declarations are as follows:

- **VAL** declarations
- **PROTOCOL** declarations
- **CHAN** declarations for channels that will be placed on transputer links ('hard' channels)
- **#USE** directives referring to separately compiled units (`.t`*xx* or `.c`*xx* files but *NOT* libraries)
- **#INCLUDE** directives referring to files containing constant and protocol declarations only.

Process code can include:

- **PLACE** statements that place channels at link addresses.
- Any other occam source except calls to libraries. These can include:
  - Declarations of constants and protocols used in the source
  - Placement of variables
  - Abbreviations and retypes of variables
  - Declarations of soft channels, that is, channels between processes on the same transputer
  - **#USE** Directives referencing any separately compiled units, except those referencing libraries

- **#INCLUDE** directives referencing any legal occam source
- Calls to separately compiled procedures and/or functions

13.5 Error messages

All errors cause the configurer to abort with an error message. Messages are in the standard toolset format, which is described in section 3.7.5.

Where the configuration description contains source code to be compiled, compiler error messages may be generated. For details of these messages see section 23.10.

Messages not in this list may be generated by corrupt files and by files not created by the toolset.

Closing code file (*filename*)

File system error. The code file *filename* could not be closed.

Closing map file (*filename*)

File system error. The map file *filename* could not be closed.

Code is not contiguous

The code being referenced was not produced by the toolset compiler, or has not been correctly linked.

Code size calculation overflow for processor

The code size limit of 2 Gbytes has been exceeded.

Configuration code is not linked correctly

The code being referenced has not been correctly linked. The most likely causes of this error are that the code has not been linked at all or has been linked with the '**U**' option.

Directive not permitted for configuration

Only **#USE** and **#INCLUDE** directives are permitted in a configuration description.

Libraries cannot be used by the configurer

You may neither use nor reference libraries directly within a configuration description.

Load address calculation for processor

The program workspace has exceeded the internal buffer limit. The limit is set at around 2 Gbytes.

Load path maximum exceeded

The buffer that holds data about the load path has been exceeded. Linear networks consume more buffer space than branched networks, and it may be possible to avoid buffer overflow by reconfiguring the network in a more linear way.

Main code too long for buffer

The main program code has exceeded the internal buffer limit. Use fewer processors in the configuration or reduce the size of the main code.

Opening code file (*filename*)

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

Opening map file (*filename*)

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

Opening object code file for extraction

File system error. An object file could not be opened.

Processor number out of range

There are too many transputer the configuration. If the statements are of the same form you may be able to use a replicated **PLACED PAR** instead. If not, you may need to reconfigure the program on a transputer evaluation board with a larger memory.

Saved code buffer overflow

A configurer software limit has been exceeded by the presence of too many processes with small work spaces. Increase the workspace requirements of the processes, or reconfigure the program on a transputer evaluation board with a larger memory.

Stack overflow

The stack holding data about the load path has overflowed. Reconfigure the program on a transputer evaluation board with a larger memory or rearrange the network in a more linear way.

Stack underflow

A configurer software limit has been exceeded. Reconfigure the program on a transputer evaluation board with a larger memory.

Too many PROCESSORs

There are too many processors in the configuration. Use a replicated **PLACED PAR** or reconfigure the program on a board with more memory.

Unable to open code file (*filename*)

File system error. A code file could not be opened.

Writing code file (*filename*)

File system error. The code file *filename* could not be written.

Writing map file (*filename*)

File system error. The map file *filename* could not be written.

14 idebug — debugger

This chapter describes the toolset debugger **idebug**. It begins by introducing the types of programs that can be debugged and explains how to invoke the tool from the operating system. It goes on to describe the symbolic debugging facilities that support source code to be examined, and the Monitor page facilities that support debugging of transputer code. Finally, the chapter outlines how some features of the debugger are implemented. The chapter ends with a list of error messages.

14.1 Introduction

The toolset debugger provides an integrated environment for debugging occam programs. Using the symbolic facilities errors can be located in source code, parameters can be examined, channels and processes can be analysed, and procedures can be traced. Using the Monitor page facilities process and timer queues can be analysed, memory contents displayed and compared, and specific portions of memory disassembled into transputer instructions. Within the Monitor page environment, symbolic facilities can be recalled to inspect variables and processes.

The debugger can be used to analyse programs running on large networks of transputers. Programs can be examined processor by processor, individual memories inspected and disassembled, and networks analysed.

14.1.1 Debugged code

An important feature of the debugger is that the compiled code is unaffected, that is, a program compiled for debugging is the same as the program compiled with debugging information disabled. Disabling debugging at compile time speeds up compilation and reduces file space requirements, but does not alter the object code that is produced.

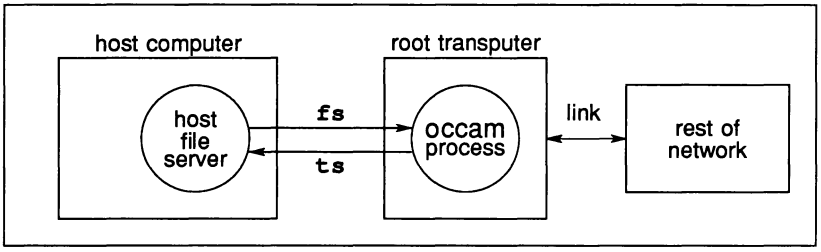
14.2 The root transputer

The debugger can be used to debug all types of single and multitransputer programs. The technique and commands to use when invoking the debugger differ slightly depending on whether or not the program (or a process forming part of the program) runs on the root transputer.

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.

The relationship of the root transputer to the host computer and the rest of the network is illustrated below.



There are two main ways of debugging programs, depending on whether or not the application is configured to use the root transputer. Command line options are used to select the two debugging modes. Programs configured to 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. R-mode programs must be debugged by first invoking the memory dump tool **idump** to save the contents of the root transputer's memory.

Before any program can be debugged the Analyse signal must be asserted once on the transputer or transputer network. Because different procedures must be adopted when debugging the two program modes, the debugger cannot assert the signal automatically.

In T-mode programs the **iserver** '**SA**' option must added to the debugger command line in order to assert the signal. In R-mode programs the **idump** tool itself asserts the signal and a second assertion would cause data in the memory to become corrupted. For R-mode programs therefore the '**SA**' is not required and should not be supplied. If **idump** is not invoked then the debugger cannot load onto the root transputer and a booting error is reported by **iserver**.

Further details about the procedures and commands to use when debugging programs can be found below. A summary of the commands to use for the two program modes can be found in Table 14.2.

14.2.1 T-mode programs

The most common way to use the debugger is to debug a complete network down a transputer link. The link is specified by the debugger '**T**' option and the **iserver** '**SA**' option is added to the command line to direct the server to assert **Analyse**.

If the '**SA**' option is not given, the debugger will not be booted onto the root transputer and the server will abort 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.

T-mode programs are loaded by using the skip tool **iskip** in conjunction with **iserver**. The skip tool allows the program to be loaded onto the network over the root processor. It activates a special route-through mechanism on the root transputer that allows the program running on sub-network to communicate with the host as though the root transputer were absent. With this mode of loading the root processor runs no program code, allowing the debugger to run on the root transputer without corrupting the program.

For details of the **iskip** tool and how to use it see chapter 22 and section 6.5.

14.2.2 Debugging R-mode programs

Code running on the root transputer is debugged from a *memory dump* file that is created using the **idump** tool. The file must be created before the debugger is invoked to debug the program. The debugger is then invoked on the root transputer's memory using the '**R**' option. Code on other transputers on the network is debugged down the transputer link.

See chapter 15 for details of how to use the **idump** tool.

14.2.3 Debugging from a network dump file

The toolset debugger is a post-mortem tool, that is, it can only analyse programs that have failed or have been externally halted. Programs cannot be re-started once the debugger has been invoked.

To suspend a 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. The debugger can then be invoked on the file without being connected to the network.

For details of the network dump facility, see section 14.5.1.

Note: Memory dump files and network dump files are not the same; the former contains a single processor's memory image, while the latter contains data about a complete network. They are also in different formats.

14.2.4 Debugging a dummy network

The debugger can also be used to debug a program using dummy data. Using the debugger option 'D' which simulates the contents of memory locations and registers, static features of a program can be examined, for example, processor connectivity and memory mapping. The dummy facility can also be used to explore the features of the debugger.

14.3 Running the debugger

Before running the debugger, ensure that the host environment variables **ITERM** and **IBOARDSIZE** are defined on the system. For details of how to set up these variables see the toolset Delivery Manual and the operating system documentation.

The example ITERM file defines a default screen size of 80 x 24 for the debugger, and a minimum screen size of 79 x 24. For a listing of the file see appendix D.

To invoke the debugger, use the following command line:

idebug *bootablefile* {*options*}

where: *bootablefile* is the name of a loadable program file. The file must have one of the extensions **.bxx**, **.btl**, or **.btr**.

options is a list of zero or more of the debugger options given in table 14.1. If no options are given, brief help information is displayed.

After the command line has been read, the debugger uses the configuration descriptor file to build a data base for the network. The complete network is then analysed, and some data retrieved from every processor. If the 'R' or 'N' options are selected, the information is read from the appropriate file.

The debugger then determines which processor (if any) has its error flag set, and will continue with that processor selected as the current processor. If no flag is set, the root processor is selected. The debugger then displays the source code in the vicinity of the error.

14.3.1 Debugging programs on B004-type boards and TRAMs

On transputer boards the **Analyse** and **Reset** signals can be propagated from the root transputer in two ways. Either the signals are propagated unchanged to all transputers on the network (wired *down*), or the signals are connected to the subsystem port (wired *subs*) from where they are controlled by the root

Option	Description
T *linknumber*	Debugs a program that does not use the root processor, on a network that is connected to link *linknumber*. *Must* be accompanied by the **iserver** 'SA' option.
R *filename*	Debugs a program that uses the root processor. *filename* is the file that contains the contents of the root processor. The file is assumed to have the extension **.dmp**.
N *filename*	Debugs a network from the network dump file *filename*. The file is assumed to have the extension **.dmp**.
A	Analyse subsystem. Directs the debugger to assert **Analyse** on the network.
D	Debugs a dummy network. Must be accompanied by the **iserver** 'SR' option.
Options must be preceded by '-' for UNIX-based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Spaces between the options and the case of letters in parameters are not significant.	
Options may be specified in any order.	
The **iserver** option 'SA' *must* accompany the 'T' option.	

Table 14.1 Debugger options

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, **Analyse** must not be asserted more than once, otherwise program data may be corrupted.

In addition, TRAM boards and B004 boards differ in the way the subsystem port is used. On TRAMs the signals are propagated to all transputers on the network, whereas on B004 boards the signals are not propagated at all.

All these conditions influence the commands to use when debugging T-mode and R-mode programs. To simplify matters, Table 14.2 lists the commands to use for different combinations of board types, subsystem wiring, and program mode.

The type of board can be identified by the hardware addresses of the three

subsystem registers. On B004-type boards the addresses are as follows:

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.

Board	Wiring	Mode	Command(s) to use
TRAM	*down*	T	**idebug** *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
	subs	T	**idebug** *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
B004	*down*	T	**idebug** *program* **-t** *linknumber* **-sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename*
	subs	T	**idebug** *program* **-t** *linknumber* **-a -sa**
		R	**idump** *outputfile size* **idebug** *program* **-r** *filename* **-a**
Modes: R = program using the root transputer; T = program not using the root transputer, and debugged down a link. Options on the **idebug** command line that are not debugger options are passed to **iserver**. For non-UNIX based toolsets use the '/' option switch character.			

Table 14.2 Commands to use when debugging B004 and TRAM boards

14.4 Debugger symbolic facilities

If a program fails because of an error, the debugger enters symbolic mode and displays the source corresponding to the error. If the program was still executing correctly when **Analyse** was asserted, as when the program is halted externally, the debugger displays the last source line executed. However, if a program fails because a transputer has stopped or deadlocked, rather than halted upon finding an error, there is no last instruction and the debugger automatically enters the Monitor page debugging environment. For details of the Monitor page facilities, see section 14.5.

While the source is being searched, the debugger displays the following message at the top of the screen:

```
Locating ...
```

If the source is in a library for which the source code is unavailable, the debugger locates instead to the line corresponding to the library call, and if necessary will continue to backtrace through the code until some source code is found to display. As this is done, the following message is displayed:

```
Backtracing ...
```

When the debugger has successfully located the source, it displays the name of the library which it first tried to display, and the name of the module displayed within that library.

If the location is in a section of normal occam source, the context of the source is displayed on the screen, and the symbolic debugging facilities become available.

Note: In certain situations the location displayed may not correspond exactly to the expected location. In particular, if no valid branch of an **IF** or **CASE** has been found, the debugger will locate to the statement *following* the construct. For more details see section 7.4.

14.4.1 Scrolling the display

The keyboard cursor keys can be used to scroll the screen display by a single line. If the terminal does not support single line scrolling the display may be slow to refresh because the debugger redraws the screen each time.

14.4.2 Compiling modules for symbolic debugging

Modules which are to be inspected symbolically should be compiled with the debugging data enabled; this is also the compiler default. Compiling a module with debugging enabled does not affect the code produced in any way; it merely controls whether the debug information is produced, and therefore whether the debugger can be used to analyse that code. No extra bugs will be introduced (or existing bugs hidden) by recompiling with debugging disabled.

14.4.3 Non-occam programs

If the module to be debugged was not written in occam symbols such as variables cannot be inspected because the compilers do not provide symbolic data

for the debugger to use.

14.4.4 Symbolic functions

The symbolic debugging functions available with the toolset debugger are listed in table 14.3. All functions are invoked using special function keys which vary from terminal to terminal. Keyboard layouts for specific terminals can be found in the rear of the Delivery Manual that accompanies the release.

Function	Description
INSPECT	Display the type and value of an occam symbol.
CHANNEL	Locate to the process waiting on a channel.
TOP	Locate back to the error, or last occam location.
RETRACE	Retrace the last BACKTRACE etc.
RELOCATE	Locate back to the last location line.
INFO	Display some extra information.
SEARCH	Search for a specified string.
LINKS	Display the link connections.
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 location of source line in memory.
CHANGE FILE	Display a different source file.
ENTER FILE	Change to an included file.
EXIT FILE	Change to an enclosing file.
GOTO LINE	Go to a specific line in the file.
TOP OF FILE	Go to the first line in the file.
BOTTOM OF FILE	Go to the last line in the file.

Table 14.3 Debugger symbolic functions

INSPECT

Note: This function is not available for non-occam source files.

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.

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... . Any letters (A to F) in a hexadecimal number must be in upper case. 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 functions available at the 'Monitor page' (see section 14.5).

Inspecting placed channels

For channel variables that have been placed into a specific memory location the INSPECT function displays the address of the location rather than its value.

Channels can be examined in detail using the CHANNEL function.

CHANNEL

Note: This function is not available for non-occam source files.

Use this key as you would INSPECT, but specify the name of a channel. Instead of displaying the Iptr and Wdesc of the process waiting for communication on that channel, the debugger locates to the corresponding line of occam source, from where you can continue debugging the process. This function is invalid if the symbol specified is not a channel.

Jumping to other processors

The CHANNEL key also allows you to 'jump' from one processor to another along hard channels (channels mapped onto transputer links). If a process is waiting for communication from the processor at the other end of the specified channel, the debugger changes to that processor. If there is no process waiting you are informed, and if the debugger is already located at the waiting process the following message is displayed:

```
Already located - No process is waiting at the
                  other end of this link
```

TOP

This function forces the debugger to locate back to the line containing the original error that crashed the program, or to the line located to by the Monitor page 'G' or 'O' commands (see section 14.5.1).

RETRACE

This function forces the debugger to locate back to the previously displayed location. Repeated use of RETRACE reverses the effect of successive BACKTRACE, CHANNEL, and TOP operations.

RELOCATE

This function relocates to the last location point. This allows you to return to the original source line after examining a section of the source code.

INFO

This function displays the **Iptr** of the last location, the corresponding **Wdesc**, in hex, the process priority, and the current processor's number and type. For example:

```
Located to Iptr #80001564, and Wdesc #80000124,
          (Hi pri), Processor 2 (T800)
```

If a **Wdesc** has not been supplied, it is given as '**invalid**'.

SEARCH

This function searches forwards for a particular string. Either specify the search

string, or press ENTER to accept the default, which is the last string searched.

LINKS

This function provides a quick means of determining the connections on the currently displayed processor. It lists each link in turn, and the processor and link to which it is connected. For example:

```
Links: L0 to host. L1 to P3 L2. L2 ---. L3 to P45
L0.
```

CODE INFORMATION

This function displays a brief summary of the debugger function keys.

MONITOR

This function transfers the user to the debugger 'Monitor page'.

FINISH

This function quits the debugger. The 'Q' option from the Monitor page has the same effect.

BACKTRACE

This function locates to the line corresponding to the call of the currently displayed procedure or function. If the current location is in the program's top level procedure, the following message is displayed:

```
Error : Cannot backtrace from here
```

GET ADDRESS

This function displays the address of the transputer code which was compiled from the current source line.

CHANGE FILE

This function opens a different source file for reading.

ENTER FILE
EXIT FILE

This function allows quick access to an occam **#INCLUDE** file. To display the file position the cursor on the **#INCLUDE** directive for the file, and press ENTER FILE.

The EXIT FILE function provides the reciprocal operation to ENTER FILE. It quits the file and displays the line containing the **#INCLUDE** directive for the file.

GOTO LINE

This function allows you to change to a particular line. Specify a line number or type 0 (zero) to abort the operation.

TOP OF FILE
BOTTOM OF FILE

These keys change to the top or bottom of the file respectively.

14.5 Monitor page

On entering the Monitor page environment from symbolic mode, the debugger displays a page of information about the current processor, including its number and type, the cause of the error, the last instruction executed, and a map of the transputer's memory.

Information also displayed on the monitor page includes internal pointers, status flags, and other low level data. This is summarised in table 14.5.

Iptr points to the last instruction executed. Low priority **Iptr** and **Wdesc** are only displayed if the processor was running in high priority mode when it was halted.

If **Wdesc** contains the value **MOSTNEG INT**, it is given as '**invalid**', and means that no process was executing on the processor when it was halted. This can occur if the processor is deadlocked; to determine the status of processes waiting for communication on the links, use the '**L**' command.

If **Wdesc** contains the address of 'Memstart' it is displayed as such. This normally means that the **Analyse** signal has been asserted on the network more than once. This may be because the host transputer board (e.g. IMS B004) has been modified to assert its subsystem signal when it is itself analysed. If this

Label	Meaning
Iptr	Contents of instruction pointer (address of the last instruction executed)
Wdesc	Contents of workspace descriptor
IptrIntSave	Contents of saved low priority instruction pointer
WdescIntSave	Contents of saved low priority workspace descriptor
Error	Whether the error flag was set
FPU Error	Whether the FPU error flag was set (if it exists)
Halt On Error	Whether the halt on error flag was set
Fptr1	Pointer to the front of the low priority active process queue
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
TPtr0	Pointer to the high priority timer queue
Clock1	Value of the low priority clock
Clock0	Value of the high priority clock

Table 14.4 Monitor page data

happens the refer to section 14.3.1 for further guidance.

An asterisk next to either **Iptr** or **Wdesc** indicates that they do not correspond to a valid code and data pointer for the program. It may be possible to find the source of the problem by using the 'M' command to display a memory map for each transputer.

14.5.1 Monitor page commands

Tables 14.5 and 14.6 summarise the Monitor page commands.

Key	Meaning	Description
A	ASCII	View a portion of memory in ASCII.
C	Compare	Compare the code on the network with the code that should be there, to ensure that the code has not become corrupted.
D	Disassemble	Display the transputer instructions at a specified area of memory.
E	Next Error	Switch the current display to data from the next processor in the network which has halted with its error flag set.
G	Goto process	Goto source level debugging for a particular process.
H	Hex	View a portion of memory in hexadecimal.
I	Inspect	View a portion of memory in any occam type (e.g. `REAL32`).
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 that transputer.
N	Network dump	Copy the entire state of the transputer network into a 'network dump' file, so that you can continue debugging later.
O	occam	Resume the occam source level symbolic features of the debugger.
P	Processor	Switch the current display to data from a different processor.
Q	Quit	Leave the debugger and return to the host operating system.
R	Run queue	Display instruction pointers and workspace descriptors of the processes on either the high or low priority active process queue.
T	Timer queue	Display instruction pointers, the workspace descriptors and the wake-up times of the processes on either the high or low priority timer queue.
X	Exit	Return to symbolic mode.
?	Help	Display a help screen.

Table 14.5 Monitor page commands

Key	Meaning	Description
RETRACE RELOCATE		As symbolic mode.
CURSOR UP CURSOR DOWN LINE UP LINE DOWN PAGE UP PAGE DOWN		Scroll the currently displayed memory, disassembly, or queue.
CURSOR CURSOR RIGHT		Scroll the currently displayed processor.
CODE INFO	Help	Display help information.
REFRESH	Refresh	Re-draw the screen.
TOP		Locate to the last instruction executed on the current processor.

Table 14.6 Monitor page commands (continued)

Full descriptions of the Monitor page commands follow. The commands are listed in alphabetical order.

A — ASCII

This command displays transputer memory in ASCII format. Specify a start address after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press ENTER 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`', which assumes the prefix `#8000....`

The memory is displayed as sixteen rows of 32 ASCII bytes. The bytes are displayed in order, with a '.' replacing any unprintable characters.

The address at the start of each line is an absolute address displayed as a hexadecimal number. The byte containing the specified start address is the top leftmost byte of the display.

CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN, PAGE UP, and PAGE DOWN keys can be used to scroll the display.

C — Compare memory

Compare memory compares the code on the network with the code that was loaded, to check that memory has not become corrupted. The following options are available:

```
      Compare memory
Number of processors in network is : 'n'

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.

The format of the display is similar to the following example:

```
No of processors checked so far : 'n'
Checking processor : 'p' ...
Bytes to test      : nnn
Checking memory    : #hhhhhhhh to #hhhhhhhh ...
Checking address   : #hhhhhhhh ...
Checked processor  : 'p' OK
Checked processor  : 'p', 'e' errors
```

When the check is complete one the following messages is displayed:

```
Checked whole network OK

'n' Errors, first at #hhhhhhhh on processor 'p'
```

Checking a single processor – option B

Option '**B**' checks just the current processor. In all other respects it is similar to option '**A**'.

Compare memory on screen – option C

Option 'C' displays the actual and expected code for for each address in a block of memory. Discrepancies are marked with an asterisk ('*').

Memory is checked in blocks of 128 bytes. At the end of each block, type either 'Q' to quit, or SPACE to read and display the next block.

The format of the display is similar to the following example:

```
                Network Code        Correct Code
#800001234 : 0011223344556677  7766554433221100 *
#80000123C : 0011223344556677  0011223344556677
#800001244 : 0011223344556677  7766554433221100 *
   ...              ...              ...
#8000012AC : AABBCCDDEEFF0011  AABBCCDDEEFF0011

Press [DOWN] to scroll memory, [SPACE] for next
error, or Q to quit :
```

Pressing SPACE automatically invokes option 'D' – `Find first error...`.

Find first error – option D

Option 'D' searches the current processor's memory for the first occurrence of a discrepancy. If a discrepancy is found the display is switched to mode 'C' and the memory can be checked and displayed as in 'Compare memory on screen'.

D — Disassemble

The Disassemble command disassembles memory into transputer instructions. Specify an address at which to start disassembly after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press ENTER 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`', which assumes the prefix `#8000....`

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 corre-

sponding 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.

CURSOR UP, CURSOR DOWN, PAGE UP, and PAGE DOWN keys scroll the display 16 bytes at a time, and LINE UP and LINE DOWN by 8 bytes at a time.

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 order in which the processors are stored in the debugger's internal data base, not in the order of processor number. 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 occam source line which caused the error.

If there is only one processor in the network you are informed:

```
There is only one processor in the network
```

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 ENTER 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', which assumes the prefix #8000.... 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.

CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN, PAGE UP, and PAGE DOWN keys may be used to scroll the display.

I — Inspect

The Inspect command can be used to inspect the contents of an entire occam array. Specify a start address after the prompt:

```
Start address (#hhhhhhhh) ?
```

Either press ENTER 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', which assumes the prefix #8000....

The start address of an array can be found using the symbolic function INSPECT. Press INSPECT while the cursor is positioned over the array name, then press ENTER when prompted for a subscript.

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 occam type (1 - INT) ?
```

Give the number corresponding to the occam type you wish to display, or press ENTER to accept the default type.

The memory is displayed as sixteen rows of data. ASCII arrays are displayed in the format used by the Monitor page command 'ASCII'. Other occam types are displayed both in their normal representation and hexadecimal format.

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.

CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN, PAGE UP, and PAGE DOWN keys can be used to scroll the display.

L — Links

The Links command displays the instruction pointer, workspace descriptor, and priority, of the processes waiting for communication on the links, or for a signal on the **Event** pin. If no process is waiting, the link is described as 'Empty'. Link connections on the processor, and the link from which the processor was booted are also displayed.

The format of the display is similar to the following example:

```
Link 0 out Empty
Link 1 out Empty
Link 2 out Iptr: #80000256 Wdesc: #80000091 (Lo)
Link 3 out Empty
Link 0 in  Empty
Link 1 in  Empty
Link 2 in  Iptr: #80000321 Wdesc: #80000125 (Lo)
Link 3 in  Iptr: #80000554 Wdesc: #80000170 (Hi)
Event  in  Empty

Link 0 connected to Host
Link 1 not connected
Link 2 connected to Processor 88, Link 1
Link 3 connected to Processor 23, Link 3

Booted from link 0
```

M — Memory map

The Memory map command displays a memory map of the current processor. The display includes the address ranges of on-chip RAM, program code, configuration code, workspace and vectorspace, the sizes of each component in bytes rounded up to the nearest 1K bytes, total memory usage, and the address of 'MemStart', the first free location after the RAM reserved for the processor's own use.

Also displayed is the maximum number of processors that can be accommodated by the debugger's buffer space. This will depend on the amount of memory on the root processor, indicated to the debugger by the host environment variable **IBOARDSIZE**.

The format of the display is similar to the following example:

```
            Memory map
Workspace           : #80000064-#800000F3 (144)
Configuration code  : #800000F4-#80000117 ( 36)
Program body        : #80000118-#80012773 (74K)
Vectorspace         : #80012774-#80024643 (72K)

Total memory usage  : 149060 bytes  (146K)

On-chip memory (2K): #80000000-#800007FF
MemStart            : #80000048

Debugger has enough memory for 1271 processors
```

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)
```

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.

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 in the following example:

```
                    Create network dump file
Number of processors : 10
File size will be    : 89673 bytes

Continue with network dump (Y,N) ?
```

To continue with the network dump, type '**Y**' and specify a filename after the prompt:

```
Filename ("network.dmp", or "QUIT") ?
```

Press ENTER 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, the following message is displayed at the terminal:

```
Dumping network to file "network.dmp" ...
Processor 99 (T800)
Memory to dump : 10456 bytes ...
```

O — occam

The occam command restores symbolic debugging, either at the same occam line, or at another location. It can also be used to debug non-occam programs. 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 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 ENTER 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**', which assumes the prefix **#8000....**

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, one of the following errors is displayed:

```
Error : Cannot locate to configuration level code

Error : This address is not in a code area
```

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 ENTER 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, the **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 variables or channels, or to backtrace, will give one of the following messages:

```
Wdesc is invalid - Cannot backtrace

Wdesc is invalid - Cannot inspect variables
```

If the location to be displayed is in a library for which the source is not available, the debugger cannot locate the call to that library, and the following message is displayed:

```
Wdesc is invalid so cannot backtrace out of
compacted 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] — Processor

This command changes to a different processor in the network. Specify the processor name after the prompt:

```
New processor number ?
```

Type a processor number (the number used to identify the processor in the program configuration description). 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 : Processor does not exist
```

If there is only one processor in the network you are informed:

```
There is only one processor in the network
```

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 much of the contents of the network.

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, the debugger displays the non-empty queue.

The screen display is paged. To view other processes scroll the display using the CURSOR UP, CURSOR DOWN, LINE UP, LINE DOWN, PAGE UP, and PAGE DOWN keys.

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.

X
RETRACE
RELOCATE — Exit

These commands return to the debugger symbolic mode. They cannot be used if you have changed processor while in the Monitor page.

CURSOR UP
CURSOR DOWN
PAGE UP
PAGE DOWN — Scroll display

These keys are used to scroll the debugger displays. The memory dump is

scrolled by eight lines (256 bytes of ASCII data, 128 bytes of hex data) the typed memory dump by sixteen lines, and the disassembly display by sixteen bytes, followed by an alignment to the start of that instruction. The memory display wraps round when the highest memory address is reached. The process and timer queues are scrolled by fourteen lines.

LINE UP
LINE DOWN — Scroll one line

These keys scroll the debugger displays by one line. The disassembly display is scrolled by eight bytes, then aligned to the start of that instruction. The memory display wraps round when the highest memory address is reached.

CURSOR LEFT
CURSOR RIGHT — Change processor

CURSOR LEFT changes to the preceding processor in the debugger's list and displays processor information. CURSOR RIGHT changes to the succeeding processor.

The sequence of processors used by the debugger may differ from the processor numbers in the configuration code. The display always shows the processor number given in the configuration code.

If there is only one processor in the network, you are informed:

```
There is only one processor in the network
```

TOP — Display last instruction

The TOP key is used to display the source corresponding to the last instruction to be executed on the current processor. The effect is the same as typing '**G**', then '**I**'.

?
HELP — Help

These commands display a page of help information, which lists the commands available at the Monitor page.

REFRESH — Refresh

This command redisplays the screen.

14.5.2 occam run time errors

This section lists some of the causes of run time errors in occam programs. The behaviour of a program when an error occurs depends on the mode in which the program was compiled, as follows:

- In HALT mode, errors halt the transputer.
- In STOP mode, errors stop the process, allowing other processes executing on the same transputer to continue.
- In UNDEFINED mode most errors are not detected.

The compiler perform many checks that might otherwise be performed at run time. For example, if an array is subscripted by a constant value, the compiler performs the range check and no extra code is inserted to perform the check at run time.

STOP In occam 2 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 no **ELSE** statement.
- **ALT** statements where no guard evaluates TRUE.

Arithmetic errors Arithmetic errors such as overflows and divides by zero cause an error.

Floating-point calculations cause an error if any input is 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 causes an error.

Type conversions Type conversions where the value is not in the range accepted 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**' disables 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 5, 6, 7 and 8 of a **BYTE** array cannot be retyped as **INT32**, since **INT32**s must be aligned on a word boundary.

PRI PARs Execution of a **PRI PAR** from within a high priority process generates an error.

14.6 Implementation notes

This section is intended for those who wish to gain some general background on how the debugger works and some features are implemented.

14.6.1 Debugging information generated by the compiler

The compiler automatically generates information for the debugger to analyse, unless the disable debug option '**D**' is specified. Debug information in object code can be displayed using the binary lister tool **ilist**.

An important aspect of the debugger is that the generation of debugging information does not alter the compiled transputer code that is produced. A program compiled with debugging enabled behaves in the same way as if debugging was disabled. Disabling debugging simply speeds up compilation and reduces file requirements.

Debug information generated by the compiler includes:

- Workspace offsets for all variables, procedure and function parameters,

abbreviations, channels, and arrays, together with their types.

- Types and values of all declared constants.
- Names of all protocols and their variant tags, together with ports and timers.
- Workspace requirements and locations of each procedure and function.

Using this information and the configuration details of the program, the debugger builds an internal map of the locations of all variables currently in use on any processor in the network.

On any one transputer, there may be many different processes executing the same portion of code. Each process has a different workspace, where local variables and channels are stored. The same code may also be executing on other processors in the network.

14.6.2 Accessing the network

The debugger is capable of analysing and debugging large networks of transputers of mixed type.

The technique used to analyse transputer networks is described in detail in *INMOS technical note 33 'Analysing transputer networks'*, and only aspects of it are described here.

To debug a network program, the debugger first builds a map of the network from the program configuration details. It then loads a special program over the network which sets up a message routing system that allows access to any occam processes running on any transputer in the network.

Before loading the routing program, the debugger first copies part of each transputer's memory into a host buffer, to make room for it. Since the program uses about 700 bytes per processor, a 10 processor network would require 7 Kilobytes of buffer space, and a 10 000 processor network would require 7 Megabytes. It is not necessary to buffer the complete memory contents of the network, because the debugger can set up communication paths to any transputer on the network. This makes it possible for the debugger to analyse and debug large networks of transputers.

Once the network has been analysed and the routing program loaded, the debugger determines the last instruction executed and the workspace requirements of each processor in the network. It uses the pointer to the last instruction and instruction pointers taken from the active process and timer queues and processes

waiting on the transputer links, to find occam processes to be examined.

occam processes are identified on the network using the transputer number, an instruction pointer, and a workspace descriptor.

14.6.3 Backtracing

The BACKTRACE function, used to trace nested procedure calls, works in the following way.

1 Using the workspace requirements and code layouts of all procedures and functions in the program, and given the instruction pointer, the debugger determines the process currently being executed and its workspace requirements.

2 From the workspace descriptor, it reads the return address of the process, and locates the procedure call.

3 The workspace is then adjusted to allow for that used by the procedure and the procedure call, to give the workspace descriptor for the calling statement.

4 The workspace descriptor for the calling statement is used together with the return address, to locate to the occam line containing the procedure or function call.

The backtrace is then complete.

14.6.4 Accessing variables and channels

The compiler produces a map for each procedure showing the workspace offset and type of each variable, parameter, or abbreviation used within that procedure. Given the instruction pointer that indicates which procedure is being executed, and the workspace descriptor for that procedure, the debugger calculates the location of the data item within the transputer's memory, and reads the variable's contents.

Non-local variables are accessed differently, using the lexical level of the enclosing procedure. (The lexical level is the nesting level of the procedure within the occam text). This is used to backtrace along the chain of procedure calls to the correct procedure's local data space, where the variable can be found.

Channels between processes executing on the same transputer are implemented by a single word in memory. This word contains either the workspace descriptor of a process waiting for communication on that channel, or a value that indicates

the process is idle. The debugger can examine a channel to determine whether a process is waiting, read the process instruction pointer and workspace descriptor, and jump directly to that process.

14.7 Error messages

Other messages not in this list may be generated by corrupt files and by files not created by the toolset.

Already located - No process is waiting at the other end of this link

Attempt to jump down a channel has failed because there is no process waiting at the other end of the link.

Cannot find correct SC (offset n) in "*filename*"

The descriptor file and file map files are inconsistent. This may be due to relinking part of the program.

Cannot find this line's location

The line shown on the screen does not correspond to executable code, so the debugger cannot display its address.

Cannot jump - Channel points to an invalid location

The contents of this channel do not point to a known process executing on this transputer.

Cannot locate to configuration level code

The address which you have asked to locate is in the configuration code, which the debugger cannot display.

Cannot locate - unknown bootstrap, so unknown memory map

The format of the bootstrap loader is unknown. The debugger cannot be used to debug programs that contain external bootstrap loaders.

Cannot open file

File could not be found.

Cannot read processor *number* **(T***xxx***)**

Processor *number* is inaccessible on the target network. Check the network topology. This error normally occurs because the debugger has failed to reset the network, or because the wrong core dump file has been specified. The failure is handled internally by the extraordinary link handling library routines.

Code file is too big: "*filename***"**

The link map and file *filename* are inconsistent.

CODE file size and memory map are inconsistent

The object file being checked does not correspond to the expected memory map.

Command line error : Duplicate debugger modes: 'D' and 'T'

Mutually incompatible options on the command line.

Command line error : Invalid target file extension: "*filename***"**

filename must be a bootable file, that is, it must have either '`.btl`' or a '`.b`*xx*' extension.

Command line error : Option T must be followed by a link number (0 to 3)

The '`T`' option requires a link number.

Command line error : No need to assert Subsystem Analyse

The '`A`' option is not required when you specify options '`N`' or '`D`'.

Command line error : This transputer link is connected to the host

The specified link is the communication link from root transputer to host, and is not connected to a transputer on the network.

Command line error : You must specify a filename

The command line syntax requires a filename.

Debug info too large (*reason*)

Size error on debug data. Use a processor with more memory or reduce the size of the code.

reason can be:

ix.tags is full
name table is full
ws.array is full

Did not expect a library "*filename*"

The debugger did not expect a library file.

Expected a library "*filename*"

The debugger expected a library file.

File has not been correctly linked: "*filename*"

File *filename* has not been correctly linked for use with the debugger.

FILE IS TOO BIG - truncated

Debugger buffer capacity exceeded. The buffer contains as much of the file as could be read before the capacity was exceeded.

Inconsistency : not enough code in object file

The object file being checked does not correspond to the expected memory map.

Incorrect format network dump file "*filename*"

The network dump file is in the wrong format, or the wrong file has been specified.

Iptr is in an alien language exception handler

The code is data, not executable code.

Iptr is not within the code portion of an alien language module

The code is data, not executable code.

"*symbol*" is not in dynamic scope

The symbol *symbol* exists in the module, but is not presently in scope. To inspect the symbol locate to a new position in the code where the symbol is in scope.

ITERM error on line *linenumber*, *message*

The debugger has detected a syntax error in the ITERM file. *message* describes the kind of error.

Not on a valid #INCLUDE line

The ENTER FILE function only works when the cursor is on an #INCLUDE directive.

Object file is too large

The object file being checked does not correspond to the expected memory map.

Only debugging tools and cursor keys are available

You have pressed a key which is not used.

READ ERROR - truncated

The debugger could not read all the file. The buffer contains as much of the file as could be read.

There is no enclosing #INCLUDE

You are not within an included file and so you cannot use the EXIT FILE function.

This address does not correspond to executable code

The address corresponds to data, not code.

Too many nested #INCLUDE files in "filename"

The maximum degree of nesting in #INCLUDE files is ten.

Too many PROCESSORs - There is only enough memory for *number*

The debugger requires more memory in order to operate on this many processors. Memory size is set by the IBOARDSIZE environment variable.

Unable to read environment variable "ITERM"

There is no translation for the ITERM environment variable which defines the screen and keyboard format.

Unknown bootstrap added, so unknown memory map

The format of the boostrap loader is unknown. The debugger cannot be used to debug programs that contain external bootstrap loaders.

Unknown core dump file format "filename"

The network dump file is in the wrong format, or the wrong file has been specified.

WARNING : memory not found in dump file

You have tried to read memory that is not stored in the core dump file.

Wdesc is invalid – *message*

The `Wdesc` you supplied in the Monitor page environment is invalid. Specify a correct `Wdesc` and retry the command.

message can be one of:

cannot backtrace
cannot inspect variables
cannot auto backtrace out of library

Wrong number of processors in network dump file "filename"

The number of processors does not correspond to the current program. The wrong network dump file may have been specified.

You cannot backtrace from here

The procedure you are trying to trace was called by the program's bootstrap routine, and cannot be accessed by the debugger.

You have changed file, so you can't find the address!

You may not request the address of a code line after you have changed the display to show a different file. Press RELOCATE before retrying the command.

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 teh normal way using the debugger tool **idebug**. It is required because **idebug** runs on the root transputer and overwrites all data and code in the 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 '#' character.

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.

15.3 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 `ilibr` — librarian

This chapter describes the librarian tool `ilibr` that collects compiled code files into a single unit that can be referenced by a program. The chapter begins by describing the command line and its options, and goes on to explain library modules, selective loading of modules, and library usage files. The chapter ends with some rules and hints for building libraries and a list of librarian error messages.

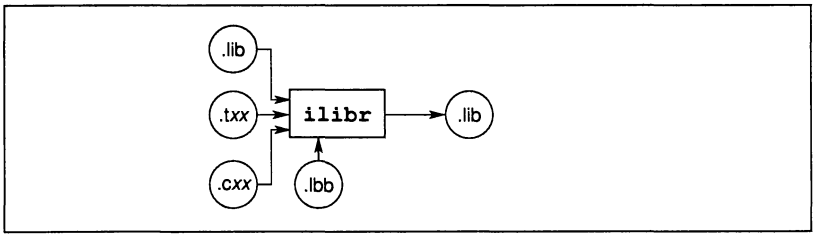
16.1 Introduction

The librarian builds libraries from one or more separately compiled units supplied as input files. The input files may be any object code file produced by the occam compiler or by any compatible compiler such as the C, FORTRAN and Pascal compilers supplied by INMOS, or files produced by the linker and librarian. In the process it enforces certain rules about the contents of libraries. Library files consist of separate modules, each originating from an input file, that are loaded as required.

16.2 Running the librarian

The librarian takes a list of compiled (and possibly linked) files, or an indirect file containing a list of such files, and integrates them into a single library file. Each file in the input list becomes a selectively loadable module in the library. Input files can themselves be library files.

The operation of the librarian tool in terms of file extensions is shown below.



To invoke the librarian type:

`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 16.1.

Option	Description
D	Disables the addition of full debugging data (includes back-trace data only).
I	Displays progress information as the program runs.
X	Explodes a library into constituent files. Also writes a library indirect file.
F *filename*	Specifies a library indirect file.
O *filename*	Specifies an output file.
Options must be preceded by '-' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Spaces between options, and the case of letters, are not significant.	

Table 16.1 Librarian options

If an output file is not specified then the library file is given the name of the first file on the command line, but with a `.lib` extension.

16.2.1 Library indirect files

Library indirect files contain a list of input files to be built into a library. To specify a library indirect file use the 'F' option followed by the filename. Each indirect file specified on the command line must be preceded by the 'F' option.

The format of library indirect files is as follows.

1 Filenames may be split over lines.

2 All characters typed on a line after the comment character '--' are ignored.

3 Options may appear on any line.

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

16.2.2 Exploding libraries into constituent files

The explode option 'X' allows a library to be disassembled into its constituent files. A copy of each file is written to the current directory. If the files already exist in the directory then they are overwritten. If an error occurs whilst the

disassembly is taking place, files that have already been created are not deleted. Exploding a library does not delete or in any other way affect the library.

The explode option can be used to change the contents of a library. For example, suppose you wish to remove a module from the library **t4lib.lib** which contains the modules **mod.t4h**, **mod.t4s** and **mod.t4u** compiled in HALT, STOP and UNDEFINED modes respectively. To remove the module **mod.t4u** you would proceed as follows:

1 Explode the library using one of the following commands:

```
ilibr t4lib.lib /x

ilibr t4lib.lib -x
```

This recreates the three constituent files **mod.t4h**, **mod.t4s** and **mod.t4u**.

2 Delete the UNDEFINED error mode file **mod.t4u** and type one the following commands:

```
ilibr mod.t4h mod.t4s /o t4lib.lib

ilibr mod.t4h mod.t4s -o t4lib.lib
```

This recreates the library **t4lib.lib** without the module **mod.t4u**.

The '**X**' option also creates a library indirect file in the current directory. The name of the indirect file is derived from the library name by adding the **.lbb** extension.

To change the contents of a library this file can be edited and used to rebuild the library.

16.2.3 Removing debug data

By default the occam 2 compiler inserts debugging data into object code. For libraries this means that the debugger is able locate to errors in the library source, just as with any other occam code, as long as the library source is present.

Debugging data can occupy large parts of the object file, and this can be undesirable in libraries. The compiler option '**D**' could be used to prevent the data being added when the files are first compiled, but this would disable debugging altogether. To allow the size of libraries to be reduced while still allowing debug-

ging, use the librarian 'D' option. This removes some of the debugging data but still permits programs that use the libraries to be debugged.

The 'D' option can also be used when a library is supplied in binary form to a third party. Although the debugger cannot locate to the library source, programs that use the library can still be debugged using the data that remains.

16.3 Library modules

Libraries are made up of one or more selectively loadable modules, each created from a single input file specified to the librarian. Modules within a library are numbered from zero for identification when the code is displayed with the binary lister **ilist**.

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

16.3.1 Selective loading

Libraries can contain the same routines compiled for different transputer types and in different error modes. The compiler and checker tools select library modules for inclusion in the program on the basis of the compilation modes of the main program. For example, if the program is compiled for an IMS T414 in HALT system mode, only modules compiled for this processor type or for a compatible transputer class, and in HALT or UNIVERSAL mode, are loaded. Rules governing the compatibility of transputer types and error modes are described in sections 23.3 and 23.4.

The linker, in the same way as the compiler and checker, selects library modules for linking according to the compilation modes of the main program. The linker links in only those library modules that are actually used by the program.

16.4 Library usage files

Library usage files describe the dependencies of a library on other separately compiled code. They consist of a list of separately compiled units or libraries referenced within a particular library. Library usage files must be given the **.liu** extension.

Library usage files should be created for all libraries that are supplied without source, so that the **imakef** tool can generate the necessary commands for linking.

Library usage files are created using use the **imakef** tool. For more details see

section 19.5.

16.5 Building libraries

This section describes the rules that govern the construction of libraries and contains some hints for building libraries.

16.5.1 Rules for constructing libraries

1 Routines of the same name in a library must be compiled for different transputer types and error modes.

2 Libraries that contain modules compiled for a transputer class (i.e. TA, TB or TC) are treated as though they contain a copy for each member of the class.

3 Libraries that contain modules compiled in UNIVERSAL mode are treated as though they contain a copy for each of the three error modes.

4 Routines of the same name in a library cannot contain the same global, static, or COMMON (FORTRAN only) variables.

5 Library source files must not contain the #SC directive.

16.5.2 Hints for building libraries

Routines that are likely to be used together in a program or procedure (such as routines for accessing the file system) can be incorporated into the same library. At a lower level, routines that will *always* be used together (such as those for opening and closing files) can be incorporated into the same module.

Try to keep separately compiled code as small as possible. This keeps the final library modules as small as possible and helps to ensure that programs do not include large amounts of code that is not used.

Libraries can contain the same routines compiled for different transputer types and in different error modes. If you compile the routines for transputer classes, or in UNIVERSAL error mode, only one copy of the code is held and the library will be smaller.

Always compile library routines with debugging enabled. This enables you to locate to the library source if an error occurs inside the library. If you wish to reduce the size of a library subsequently, for example if you are supplying a

library to a third party in binary format, use the librarian 'D' option to remove the source location data.

If a library source file references compiled code (including other libraries) then the referenced code should be included in the library. If this is not possible then the name of the code file should be included in the library usage file (see section 16.4). An alternative is to link the referenced code with the calling code before building the library,, or combine the libraries together using the librarian.

16.6 Error messages

Errors cause the librarian to abort with an error message. Messages are in the standard toolset format. If an error occurs the librarian aborts and no library files are generated.

Bad format, *reason*

Bad input file format. Check that the file is of the correct type for the librarian and that the file has not been corrupted.

reason can be:

incompatible library file
multiple IDs
not a library file
SYMBOL before TARGET

Expected file name after O option

Command line error. No output file was specified.

Illegal execution mode (*value***)**

Bad input file. *value* represents the illegal error mode.

Illegal link data tag (*value***)**

Bad input file. *value* is the illegal tag.

Illegal operand data tag (*value***)**

Bad input file. *value* is the illegal tag.

Illegal option (*char*)

Command line error. *char* is the invalid option character.

Illegal processor type (*value*)

Bad input file. *value* represents the illegal processor type.

Illegal string length (*value*)

Bad input file. *value* gives the illegal length.

Indirect file name too long (*value*)

Command line error. *value* is the number of characters read. The maximum length for library indirect files is 255 characters.

Input file name too long (*value*)

Command line error. *value* is the number of characters read. The maximum length for input files is 255 characters.

Multiple output files specified (*value*)

Command line error. *value* gives the number of files specified.

No input file specified

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

Output file name too long (*value*)

Command line error. *value* is the number of characters read. The maximum length for output files is 255 characters.

Ran out of heap memory

There is not enough memory available to run the tool. This may occur if the libraries are very large. Use a transputer board with more memory.

Symbol multiply defined, *symbol* (*target*,*mode*) in *filename*

More than one routine with same name (given by *symbol*), target processor (given by *target*), and compilation error mode (given by *mode*) was found. *filename* gives the name of the file in which the name was first

found. The file name given as part of the standard toolset error message indicates in which file the second occurrence of the name was found.

Unable to close (*value*)

File system error. The file specified in the message prefix could not be closed. *value* is the error value returned by the host system.

Unable to open (*value*)

File system error. The file specified in the message prefix could not be opened. *value* is the error value returned by the host system.

Unable to read (*value*)

File system error. The file specified in the message prefix could not be read. *value* is the error value returned by the host system.

Unable to write (*value*)

File system error. The file specified in the message prefix could not be written. *value* is the error value returned by the host system.

Unexpected end of input

Bad input file format. Check that the file type is compatible with the librarian and has not become corrupted.

X switch and D switch are incompatible

Command line error. Options '`x`' and '`D`' cannot be used together on the same command line.

17 `ilink` — linker

This chapter describes the linker tool **`ilink`** which creates single object files from separately compiled code and libraries. The chapter begins with an introduction to the tool, explains some features of its use, and describes the linker command line and its options. The chapter ends with a list of linker error messages.

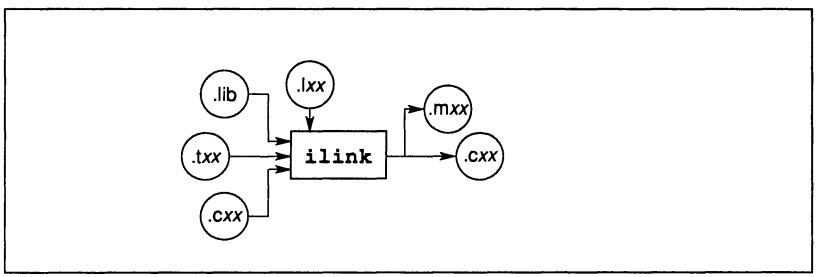
17.1 Introduction

The linker links compiled code into a single object file, resolving all external references. Code files can be separately compiled program units or library files. Code produced by the linker can be used as input to the compiler, the configurer, the bootstrap tool, the librarian, and the linker.

The linker can be driven directly via the command line or indirectly from a linker indirect file or standard input. Input from linker indirect files or standard input is known as *redirected* input.

The linker can also produce an output file in which external references are passed through to be resolved during a later linking operation. This enables sub-components of a program to be prelinked before linking the main program, and supports modular development of programs.

The operation of the linker in terms of file extensions is shown below.



17.2 Running the linker

To invoke the linker use the following command line:

`ilink` {*filenames*} {*options*}

where: *filenames* is a list of files generated by the occam 2 compiler, by INMOS-compatible compilers, by the librarian, or by the linker. If the 'O' option is not specified the name of the first file in the list is used to generate the output file. If the first file in the list was generated by the linker (extension .c*xx*) then an output file should be specified to avoid overwriting. The format for standard input is the same as for linker indirect files.

options is a list of any of the options given in table 17.1.

Option	Description
A	Displays buffer usage of linker during linking.
E	Extends linker capacity (two pass operation).
I	Displays brief information as the linking proceeds.
L	Loads the linker onto a transputer board.
M	Disables the link map. The default is to produce a link map. The file map is given the name of the first input file, but with a .m*xx* extension.
S	Creates a symbol table with a name derived from the first input file. The file is given the .s*xx* extension.
U	Allows unresolved external references. Used for prelinking program components.
V	Displays detailed (verbose) information as the linking proceeds.
B (*size*, ...)	Redefines linker internal buffer sizes. *size* is specified in decimal.
Q (*symbol*, ...)	Optimizes library functions by placing them low in memory.
O *filename*	Specifies output file.
F *filename*	Specifies a linker indirect file. If no filename is given input is taken from 'standard input'.
Options must be preceded by '–' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Spaces between the options and the case of letters in parameters are not significant.	
Options may be specified in any order.	

Table 17.1 Linker options

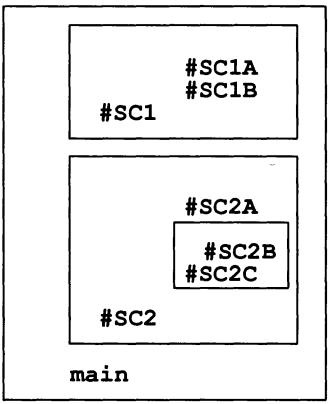
17.2.1 Ordering of input files

occam object files can be linked in any order, but because the processor type and error mode for the linked output file are determined from the first input file in the list, the main body of a unit should be specified first. This sets the execution error mode and transputer type of the linked code to that of the main unit.

If a program references separate compilation units by the #SC directive then linking must be performed in a specific order. This is because the compiler inserts the call directly, rather than allowing the linker to patch calls, and there may be conflicting entry point names in different compilation units.

The correct order, after the main body, is the reverse of the order in which the #SCs appear in the program.

For example, consider the program structure illustrated below:



The correct linking order in this program is as follows:

```
main+SC1+SC1A+SC1B+SC2+SC2A+SC2B +SC2C
```

17.2.2 Renaming entry points

To rename the entry point name for an input object file, prefix the input object file with '*newname*='. This instructs the linker to change the first entry point name in the associated input object file to *newname*.

17.2.3 Using imakef to simplify linking

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.

The **imakef** tool can be used to generate a linker indirect file. This can be useful for identifying library usage within a program and to simplify the linker command line.

17.2.4 Input files referenced by #SC

If occam object files are referenced by the #SC directive within an input file, then the input file must be prefixed with the symbol '=' before being presented to the linker. This prevents entry points defined in the file being referenced by other input object files, and instructs the linker to link files in the order specified on the command line. If you use the Makefile generator **imakef**, the necessary prefixing is done for you in the linker indirect file.

Note: Most occam programs written using the toolset do not need to use the #SC directive. You are recommended instead to use the #USE directive, which does the same job and has none of the side-effects. For more information about the #USE and #SC directives, see section 23.6.

17.2.5 Linker output

The name of an output file can be specified using the 'O' option. If no output file is specified then the file is given the name of the first file in the list, but with a *.cxx* extension. Where the first file in the list is a linker output file, that is, it already has a *.cxx* extension, an output file must be explicitly specified, or the input file may be overwritten.

The processor type and error mode for the output file are determined by the first input file on the command line that is not a library file. If any input file in the list is incompatible with the the mode that is set, the link fails and an error is reported. Library modules found to be incompatible are ignored.

If an error occurs during execution of the linker, or during any prelinking operation, any output files that have already been created are deleted.

17.2.6 Linker indirect files

Linker indirect files can contain a list of input object files, plus linker options. The format for the linker command script in the indirect file is the same as the normal command line syntax, except that input file names and options can be split over a number of lines.

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

Redirected input, such as linker indirect files, may not itself be redirected. This means that indirect files may not refer to other indirect files or to standard input.

The Makefile generator tool **imakef**, described in chapter 19, generates linker indirect files.

17.3 Features of the linker

This section describes some special features and capabilities of the toolset linker, including selective linking of libraries and prelinking of program components.

17.3.1 Selective linking of libraries

Libraries that are compiled for processor types or error modes incompatible with other files in the input list are ignored by the linker. This allows selective loading of library modules based on processor type and error mode because if libraries are available in several compilation modes, then only the required module is used.

Libraries are also selected for linking on the basis of previous usage. Library modules that are used by more than one input file are linked in only once.

See section 4.8.1 for more details about selective loading.

17.3.2 Prelinking of program components

Subcomponents of the program can be linked together prior to linking with the main program. This is known as prelinking.

Prelinking of program files produces self-contained code units in which all external references are resolved, ready for linking into a main object file. It can also be used to link object files where there may be a conflict of names.

Prelinking is necessary for program modules written in C, FORTRAN, or Pascal. These should be prelinked to form linked subunits, which can later be integrated with the main occam program.

Note: It may not be possible to use the debugger on incompletely linked files; the debugger must be able to trace the matching occam source.

Prelinking can also be performed implicitly on the command line by linking the components together syntactically. This is described in the following section.

17.3.3 Command line prelinking

To force the prelinking of a group of files on the command line, enclose the list of files in parentheses. The linker first links all the sub-components together to form a single unit, then links all the units together to form the main program.

The syntax for prelinking on the command line is illustrated below.

ilink *mainprogram* **(***subprog1 subprog2 ...***)**

Entry points defined by input object files within a group of files that are prelinked can only be referenced by files in the same unit. The exception to this is the main entry point for the unit (defined by the first non-library file in the group), which must be called by the main program.

If any symbolic references cannot be resolved within a group of files, an attempt is made to resolve them using symbols defined outside the group. This is equivalent to invoking the linker with the '**U**' option.

17.4 Linker options

This section describes the functions provided by other linker options.

17.4.1 Extending linker capacity – option E

Normally the linker makes a single pass over the input files, processing all code and link data in memory. This restricts the amount of code which can be linked. The '**E**' option forces two passes over the code, and allows larger pieces of code to be linked. Two-pass linking takes longer to complete.

17.4.2 Permit unresolved references – option U

The linker normally resolves all external references, and any unresolved external references are reported as errors.

Sometimes it is desirable to allow unresolved external references, for example when linking a sub-component of a program. The 'U' option prevents unresolved references from generating errors, and allows linking to proceed to completion. Warning messages are still generated.

17.4.3 Disabling the link map – option M

Option '**M**' disables the production of a file containing a map of the code being linked. This map is required by the debugger tool **idebug** and the simulator tool **isim**. It contains information about the order of linking of input object files, and the address range of each file's code within the overall linked code. Addresses are displayed as byte offsets from the start of the code. The map contains information about two categories of input file; separate compilation units, and library modules.

17.4.4 Symbol table – option S

This option enables the production of a symbol map. The map file is given the same name as the output file but with a **.s***xx* extension. The default if you do not specify this option, is not to produce a symbol map file.

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

17.4.5 Changing buffer sizes – option B

This option modifies the sizes of the linker internal buffers. Buffer sizes can be displayed by using the linker '**I**' option and buffer sizes can be examined using the linker '**A**' option.

There are seven internal buffers used by the linker, five of which can be modified. The modifiable buffers are LINK, DESC, SYMBOL, INIT, and REF. Two other buffers, FILE and CODE, are fixed and cannot be modified.

New sizes for the modifiable buffers are specified in parentheses and must be listed in the order:

LINK, DESC, SYMBOL, INIT, REF

To leave a buffer unchanged, leave the position blank.

For example, the following commands modify the LINK and SYMBOL buffers only, leaving others unchanged:

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

The following command modifies all buffers that can be changed:

```
B (100000,60000,5000,10000,3500)
```

The modifiable buffers are described below.

LINK This buffer holds link information from all the object files and is used by the linker to perform any linking operations specified in these object files.

DESC This buffer holds the environment information present in the object file that defines the main entry point of the program. The information contains the formal parameter specification of the main entry point, the entry point offset, and the scalar and vector work space requirements for the main entry point. Also stored in this buffer are the program's processor type and error mode, and the program's total code size.

SYMBOL This buffer stores all the code and data global symbols defined in the program being linked, including any symbols associated with COMMON blocks. It is used by the linker as its symbol table. The SYMBOL buffer uses an internal SYMBOL NAME buffer to store symbol names. The SYMBOL NAME buffer cannot be changed by the programmer but is modified whenever the SYMBOL buffer is changed. If a message is displayed indicating overflow of the SYMBOL NAME buffer the size of the SYMBOL buffer should be increased.

INIT This buffer stores information about non-occam components in a program. This information is used to initialise static variables and data segments at run-time.

REF This buffer stores all the external references that are made in each object file being linked (the buffer is re-used for each object file and library module).

Of the non-modifiable buffers, FILE contains a list of the modules specified on

the **ilink** command line, and CODE is allocated last, using whatever space is not already occupied by other buffers. FILE uses an internal FILE NAME buffer to store file names. The FILE NAME buffer cannot be modified.

Buffer sizes

Default buffer sizes are given in the following table.

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

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

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

If a specific buffer is made very large, the sizes of other buffers should be reduced accordingly, otherwise there may be insufficient memory space to allocate the CODE buffer, and the linker will report an error.

Current buffer sizes can be displayed using the linker **I** (Information) option.

Calculating memory requirements

To determine the buffer requirements of the linker in bytes, multiply the buffer sizes by the appropriate element size, and sum the results. For example:

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

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

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

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

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

where: *size* is either the default proportional setting or a value set by the '**B**' option.

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

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

Some sample buffer sizes for different values of TOTAL.SIZE which generate usable CODE buffer sizes are shown in the following table:

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

17.4.6 Optimise symbols – option Q

This option optimises the placement of specific library functions. The functions specified are placed at the front of the linked code, where they are most likely to be placed in on-chip RAM. Where functions are too large to be placed on chip, they are placed low in the address space, which on some transputer boards corresponds to the fastest off-chip RAM.

The functions to be optimised are specified using the '**Q**' option followed by a list of the corresponding entry point names. To specify a list of entry names, separate each name by a comma. Entry names can contain any character except a space or a comma and must be specified on a single line. If a specified entry name is not used by the program then no optimisation is performed for that function.

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

and division on transputers without hardware support for floating point operations.

17.5 Error messages

Other messages not in this list may be generated by corrupt files and by files not created by the toolset.

Attempted to re-redirect input

Command input has already been redirected using the '**F**' option. The '**F**' option has been given more than once on the command line, for example, by specifying it in a linker indirect file.

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

A code patch specified by an **INSTRUCTION** record has overwritten some valid code (code not composed of NOP transputer instructions). This error generally only happens when using C, FORTRAN or Pascal language inserts. *value* is the code patch offset specified by an **INSTRUCTION** record that generated the error.

End module expected

Parentheses used to group together a set of object files are not correctly paired.

End module unexpected

Parentheses used to group together a set of object files are not correctly paired.

Error mode incompatible (*value*)

A module has been compiled for an error mode incompatible with the main program. *value* is the incompatible error mode.

Expected end of buffer list

The closing parenthesis was omitted when using the '**B**' option.

Expected end of symbol list

The closing parenthesis was omitted when using the '**Q**' option.

Expected start of buffer list

The opening parenthesis was omitted when using the 'B' option.

Expected start of symbol list

The opening parenthesis was omitted when using the 'Q' option.

File name expected

This error can occur if the filename is omitted when using the 'O' option, or when prefixes '=' and '*newname*=' are used.

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

A negative or zero buffer size was specified with the 'B' option. *buffertype* can be LINK, DESCRIPTOR, SYMBOL, INIT, or REF.

Illegal character in buffer list (*char***)**

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

Illegal option (*char***)**

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

Illegal symbol offset, *symboltype (offset)*

A negative offset was found in the object code. This can be generated if a procedure uses a separately compiled unit that is declared outside the procedure body. No action is taken by the linker, which reports the error as a warning.

symboltype can be: CODESYMB, ENTRYSYMB, NEWENTRYSYMB, ENTRY, or NEWENTRY.

Illegal symbol offset, *symboltype (symbol)*

A negative offset was found in the object code. This can be generated if a procedure uses a separately compiled unit that is declared outside the procedure body. No action is taken by the linker, which reports the error as a warning.

symbol can be: CODESYMB, ENTRYSYMB, NEWENTRYSYMB, ENTRY, or NEWENTRY.

symbol is the symbol name.

Internal buffer overflow, *buffertype*

This message is generated if one of the linker buffer overflows. *buffertype* can be LINK, DESC, REF, INIT, SYMBOL, FILE, CODE, STRING, SYMBOL NAME or FILE NAME. Buffers LINK, DESC, REF, INIT, and SYMBOL can be increased by using the linker '**B**' option, CODE can be increased by adjusting the other buffers to allow for it, and SYMBOL NAME overflow can be avoided by increasing the size of SYMBOL.

The buffers FILE, FILE NAME, and STRING cannot be modified. If these buffers overflow, then the number of files to be linked, the length of the file name, or the length of character strings must be reduced. STRING has a capacity of 255 characters, and the size of the FILE buffer can be displayed using the linker '**V**' option.

Multiple entry points, *symboltype* **unchanged**

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

Multiple MAININIT addresses, MAININIT

More than one **MAININIT** operand to a record was specified. This error can occur when the run time library for non-occam programs has been specified more than once.

No MAININIT address, INIT

No **MAININIT** operand to a record was specified. This error can occur when C, FORTRAN or Pascal code is used and no run time library is specified.

Output file redefined

The 'O' option was specified more than once.

Processor type incompatible (*value*)

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

Program entry point not found

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

Reference to undefined symbol, REF (*symbol*)

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

Seen SC file, expected entry file

A non-library object file was specified using the '=' prefix before the main body for the program was specified.

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

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

Start module unexpected

Parentheses used to group together a set of object files are not correctly paired.

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

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

Unable to allocate buffer, CODE (*value*)

The CODE buffer could not be allocated. *value* is the invalid size of the buffer, or the amount of space remaining from which to allocate the buffer. This error may occur if large buffer sizes are set using the 'B' option.

Unable to close (*value*)

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

Unable to open (*value*)

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

Unable to read (*value*)

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

Unable to write (*value*)

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

18 `ilist` — binary lister

This chapter describes the binary lister tool **ilist**, which takes an object file and displays information contained in the object code in a readable form.

18.1 Introduction

The binary lister tool **ilist** reads an object code file, decodes the information, and displays useful data on the screen. The output may be redirected to a file or to some other tool such as a sort program. Command line options control the type of data displayed.

The **ilist** tool can decode and display object files produced by the compiler, linker, and librarian tools, and code generated by compatible sequential language compilers such as the INMOS C, FORTRAN and Pascal compilers.

Object code files reflect the modular structure of the original source. Each separately compiled unit in the source becomes a separate module in the object code. Single unit compilations produce a one module file, whereas compilations involving several independently compiled units, such as libraries, produce object files that contain as many modules as there were separately compiled units. The data produced by **ilist** reflects the module composition of object files.

18.2 Data displays

The binary lister can display the following types of data about object code:

- 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 information can only be provided for code produced by the toolset occam compiler.
- Entry point data – entry point names in each module, along with the target processor, compilation mode and workspace requirements of each entry point. Information is displayed in tabular form.
- External references – names of all external routines used by each module. Information is displayed in tabular form.
- Module data – data for each module including compiler version, target processor, compilation mode, and the code size.
- Tag data – all *data* contained in the object file.

- Debugging data – full debug data as generated by the occam compiler for use with the debugger and simulator.
- Code dump data – addresses and code contained in each module, in hexadecimal format.
- Global data – global variables and COMMON data produced by C, FORTRAN and Pascal compilers. Data is displayed in tabular form.

In all displays the '*' character at the end of a name indicates that the name has been truncated.

18.3 Running the binary lister

To invoke the binary lister tool use the following command line:

ilist *filename* {*options*}

where: *filename* is the object file to be displayed.

options is a list of one or more of the options given in table 18.1.

The **ilist** tool may be used to display any object code file produced by the occam 2 or any compatible compiler (**.t***xx* files), the linker (**.c***xx* files), or the librarian (**.lib** files).

If the file name is omitted and no options are supplied, **ilist** provides brief help information. If only a file name is supplied, **ilist** returns procedural interface data only.

Any or all of the options may be used, in any order. The data for each option is output in turn (the data are not mixed). The order of displays is same as the order in table 18.1.

The **ilist** tool 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.

Options and their displays are described in the following sections.

18.4 Procedural interface data – option P

This output takes the form of mock occam procedure or function declarations listing the parameters for each entry point in the object code, along with the

Option	Description
P	Displays procedural interface data. This is the default and may be omitted when only this data is required.
E	Displays entry point data.
X	Displays external reference data.
M	Displays module data.
T	Displays file tokens (tag data).
D	Displays debugging data.
C	Dumps code in hexadecimal.
V	Displays global data.
I	Displays progress information as the tool runs.
S (*n1,n2 ...*)	Selects modules to be displayed. A maximum of twenty modules may be specified.
O *filename*	Specifies an output file.
Options must be preceded by '–' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Spaces between the options and the case of letters in parameters are not significant.	
Options may be specified in any order.	
The S option restricts output of modules to those which are specified. If non-existent modules are specified they are ignored. Modules are numbered from zero.	

Table 18.1 Binary lister options

names and direction of channels used within the procedure.

An entry point corresponds to a procedure or function call available within that module, via a #USE or #SC directive. Each module is identified by its name and number and entry points can be related directly to their corresponding modules.

The module is the smallest unit of code that can be separately loaded. If only one entry point in a module is required, the code for all entry points in that module are loaded.

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.

Procedural interface data can only be displayed for code produced by the occam compiler supplied with the toolset.

An example of the display of procedural interface data from a library is shown below.

```
Library module 65 : wint64.tbu

PROC so.fwrite.int64(CHAN OF SP fs,CHAN OF SP ts,VAL INT32 streamid,
 VAL INT64 n,VAL INT width,BYTE result)
  SEQ
    fs?
    ts!
:
PROC so.write.int64(CHAN OF SP fs,CHAN OF SP ts,VAL INT64 n,VAL INT width)
  SEQ
    fs?
    ts!
:
```

18.5 Entry point data – option E

This output is displayed in tabular form. It consists of a list of entry points with their corresponding modules, along with the transputer type, compilation mode and workspace data for each entry point. The data is displayed in columns.

ASCII sort programs can be used on this output to sort the entry points into alphabetical order.

The data displayed is described below.

Entry point	Name of entry point
Module name	Name of module in which entry point is found
No	Number of module in which entry point is found
TT	Transputer type for which code is compiled
EM	Compilation error mode
Offset	Offset in bytes of entry point from start of module
Wspace	Work space requirement for entry point in machine words
Vspace	Vector space requirement for entry point in machine words

The first twenty characters of entry point names and module names are displayed. If a name is truncated in the display it is marked by an asterisk '*****'.

The transputer type **TT** is indicated by the model number of the transputer (for example **800** for the IMS T800 transputer). The compilation mode is indicated by a single character: **H** for HALT system mode, **S** for STOP on error mode, **X** for UNIVERSAL mode, and **U** for UNDEFINED mode. If either the transputer type or compilation mode is unknown an asterisk is displayed in the appropriate column.

Some implementations do not support separate vector space or workspace, in which case the **Vspace** and **Wspace** columns are left blank.

An example of the display of entry point data from a library is shown below.

```
     Entry Point       Module Name    No  TT EM  Offset  Wspace  Vspace
so.fwrite.int64    wint64.tbu         65   B  U      20      25     134
so.write.int64     wint64.tbu         65   B  U     121      35     134
so.fwrite.int64    wint64.t8h         66 800  H      20      25     134
so.write.int64     wint64.t8h         66 800  H     129      35     134
```

18.6 External reference data – option X

This output is in a tabular form. It consists of a list of modules and their external references. External references are references to separately compiled units.

The data displayed is summarised below.

No	Number of module in which entry point exists
Module name	Name of module in which entry point exists
External references	Names of separately compiled units

All modules in the file are listed in the display. If a module uses no external units the external references field is left blank. External references are displayed in two columns.

The first twenty characters of module names and the first thirty characters of external reference names are displayed. If a name is truncated in the display it is marked by an asterisk '*'.

An example of the display of external reference data from a library is shown below.

```
No    Module Name         External References

65 wint64.tbu          INT64TOSTRING          sp.write
66 wint64.t8h          INT64TOSTRING          sp.write
```

18.7 Module data – option M

This output displays data for individual modules in the object file. Data displayed includes:

- The compiler version
- The transputer type and error mode
- Comments inserted by the #COMMENT compiler directive
- The size of the code
- The size of any nested code
- The size of any debug data.

The code size data allows you to determine how much code would be loaded when a specific module is referenced. If any part of a module is required, the code for all of that module must be linked.

Data is displayed in separate blocks for each module. Some of the data is also used by other tools in the toolset.

The format of the display for each module is as follows:

The display is headed by the module name and number. This enables you to identify specific entry points.

The first one or two lines of the data display contain compiler version information, which can be used by some tools in the toolset for compatibility testing.

The next line gives the transputer type and error mode. If either of these is unknown an asterisk is displayed. Any comments introduced into the code by the #COMMENT directive are displayed after the transputer type and error mode.

The next line of the display gives the total code size for the module in bytes, and the size of any nested code, also in bytes. Nested code refers to separate compilation units that are referenced by the #SC directive.

The final line of the display gives the size in bytes of debug data in the module.

An example of the display of module data from a library is shown below.

```
Library module 65 : wint64.tbu

occam 2 product compiler (19th September 1988)
Toolset transputer compiler/configurer V1.42, 7 November 1988
Target processor : TB Compilation mode : undefined
Occam toolset i/o library, V0.00, 24 Oct 88
(C) Copyright INMOS Ltd, 1988
Code for this module : 144 bytes, Nested code : 0 bytes
Debug data for this module : 664 bytes


Library module 66 : wint64.t8h

occam 2 product compiler (19th September 1988)
Toolset transputer compiler/configurer V1.42, 7 November 1988
Target processor : T800 Compilation mode : halt on error
Occam toolset i/o library, V0.00, 24 Oct 88
(C) Copyright INMOS Ltd, 1988
Code for this module : 152 bytes, Nested code : 0 bytes
Debug data for this module : 664 bytes
```

18.8 Tag data

This output lists all *data* found in the object code file. This includes all the data described in the other output displays.

Tag data is not displayed in a tabular form but is output in the sequence in which it is found in the object code. Debugging and code data are not displayed directly, but their position in the code is indicated.

The *tag* display is intended for diagnostic support and analysis; large amounts of data are produced which may require skilled interpretation.

Four parameters are displayed for each tag. The display sequence display is as follows:

address tag type data

where: *address* is the address of the tag relative to the start of the file

tag is the name of the tag

type is the tag type

data is the data associated with the tag.

18.9 Debugging data – option D

This output displays all debugging data in the code, generated by the occam compiler. If the compiler was invoked with the 'D' option, no debugging data will

be present in the code.

For library code, the amount of debugging data added to the code when the library is built can be reduced by invoking the librarian with the 'D' option.

Debugging data is output in the order that it is found in the code. Each piece of data is preceded by an index giving the correct order of the debugging data. To put the data into the correct order, run a sort program using this index as the sort tag. Only one module at a time can be sorted in this way.

Object code can contain a great deal of debugging data, much of which may require skilled interpretation.

18.10 Code dump data – option C

This output 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 (.).

Code produced by C, FORTRAN and Pascal compilers may not be stored contiguously, or in the correct order. For non-occam code the same format is adopted, but addresses may not be displayed in the correct order and blocks of code may be broken up.

To order the code correctly, run a sort program using the address as the sort tag. Only one module at a time can be sorted in this way.

An example of the code dump output is shown below.

```
Library module 0 : OCCAM LIBRARY flibs.occ

00000000 60BF40D0 107241F7 70C4A573 7244F702  `.@..rA.p..srD..
00000010 21F5B122 F060BF72 44FE7273 FFB122F0  !..".`.rD.rs..".
```

18.11 Global data – option V

This option displays a list of global variables and FORTRAN COMMON data for specific modules, in tabular form. Data displayed include the transputer type, error mode, type of data (Global or COMMON), and offset or size of the code.

Global variables are variables that are shared between modules. COMMON data is data defined by the FORTRAN COMMON statement. Module names can be sorted into alphabetical order using ASCII sort programs.

The data displayed and their meanings are summarized below.

Variable	Name of the global variable or COMMON data.
Module name	Name of module where the data is found.
No	Number of module where the data is found.
TT	Transputer type for which the code is compiled.
EM	Error mode for which the code is compiled.
Type	Either GLOBAL for a global variable or COMMON for COMMON data.
Offset/Size	Offset of the global variable from the start of the module's static data area, or the size of COMMON data.

Only the first 20 characters of variable and module names are displayed. Where a name has been truncated this is indicated by a '*' character.

An example of the global data output is shown below.

```
     Variable         Module Name    No  TT EM    Type   Offset/Size
f67890            test.bin            0 414  *   COMMON          1000
highest_index     display.bin         1 414  *   GLOBAL             2
debug_version     display.bin         1 414  *   GLOBAL             3
```

18.12 Error messages

Bad format object file *text*

Bad object file format. The file may not be an object file, or may have

been truncated. *text* gives further technical information about the error.

Bad module parameter *chr*

Command line error. An invalid parameter was supplied with the 'S' option. The invalid parameter is given by *chr*.

Bad parameter option *chr*

Command line error. An unknown option *chr* was supplied on the command line.

Cannot open file for reading

File system error. The file could not be opened for reading. Check that the file exists.

Cannot open file for writing

File system error. The file could not be opened for writing. Check that the file exists.

Module specifier is not a number *chr*

Command line error. When using the 'S' option modules must be specified using the module number.

No file name provided

Command line error. No object file was specified.

No opening bracket in module specifier

Command line error. When using the 'S' option the list of module numbers must be in parentheses.

Only one output file may be specified

Command line error. The 'O' option can be used only once on the command line.

Selected modules already specified

Command line error. The modules have been specified more than once on the command line.

Too many modules

Command line error. A maximum of 20 modules can be specified after the 'S' option.

Too many parameters

Command line error. The command line can have a maximum of 512 characters.

While moving to start of file

A file system error occurred while the lister was trying to locate the start of the file.

Writing file

File system error on the host when writing the file.

19 imakef — Makefile generator

This chapter describes the Makefile generator **imakef** that builds Makefiles for program version control. It outlines the purpose of MAKE programs and Makefiles, and explains how to invoke the **imakef** tool and its options. The chapter ends with a description of the format of Makefiles created by the **imakef** tool and a list of error messages.

The chapter also describes how **imakef** is used to create library usage files.

19.1 Introduction

The occam 2 toolset allows you to build complex programs using many source, header and object files, and each time any part of the program is changed the whole program must be rebuilt. For programs of any size the effort required can be considerable.

MAKE programs automate the process by determining which parts of the program have changed since the last compilation and rebuilding them using information provided in a **Makefile**. Makefiles contain textual descriptions of the dependencies of files on other files, along with the command strings needed to rebuild the program or program component.

The **imakef** tool assists with the process by creating Makefiles for code generated using the occam 2 toolset. It uses the structured design of toolset compilation to reconstruct the operations required to generate all types of object and bootable files that can be generated by the toolset.

Makefiles produced by **imakef** are compatible with the following Make programs:

- Borland **MAKE**.
- Unix **MAKE**.
- GNU **MAKE**.

Many other MAKE programs can also be used with **imakef**. However, Microsoft MAKE is not compatible.

MAKE programs are freely available as public domain software and are not provided with the toolset. A MAKE program suitable for use with the toolset can be

obtained free of charge from INMOS distributors.

The source of the **imakef** tool is supplied so that it can be modified for specific systems and MAKE utilities.

19.2 What is Make?

MAKE programs provide a way of automating the recompilation process during program development. Using dependency information supplied in a Makefile in conjunction with the modification time of files MAKE is able to determine where changes have occurred and take the action necessary to rebuild a program.

19.2.1 Makefiles

Makefiles are the input files for MAKE programs. They contain information about file dependencies along with commands needed to rebuild modules or programs.

The following example shows a single Makefile entry generated by **imakef**. Examples are shown for UNIX based and non-UNIX based toolsets.

```
filer.t4h : filer.occ filer.inc userio.lib
  $(OCCAM) filer -t4 -h $(OCCOPT) -o filer.t4h
```

```
filer.t4h : filer.occ filer.inc userio.lib
  $(OCCAM) filer /t4 /h $(OCCOPT) /o filer.t4h
```

The first line identifies the files on which **filer.t4h** depends, namely the source file **filer.occ**, a file of constants called **filer.inc**, and the library file **userio.lib**. The second line represents the command that will recompile the object file **filer.t4h**. This is achieved using calls to macros defined earlier in the file. In this example the macros are: OCCAM, which represents the command that invokes the compiler; and OCCOPT, which is a list of compiler options. Macros can be redefined by editing the Makefile.

Any of the files on which **filer.t4h** depends may also depend on other files. For each of these files the Makefile will contain a set of file dependencies along with the commands needed to recreate the file.

19.3 Running the Makefile generator

The **imakef** program takes as input a list of files generated by tools in the occam toolset, and generates a Makefile for each of the input files. Each

Makefile is given the name of the first part of the object file name (no extension is added), unless an output file is specified on the command line. If a linking operation is required as part of the build, a linker indirect file is also created containing the necessary linker command script.

The syntax of the **imakef** command is as follows:

imakef {*filenames*} {*options*}

where: *filenames* is a list of files for which Makefiles are to be generated. If the file to be created is a library usage file, only one file is accepted.

options is a list, in any order, of one or more options from the following table.

Option	Description
I	Displays progress information as the tools runs.
o *filename*	Specifies output filename. The default is to use the name of the object file. No extension is added.
Options must be preceded by '–' for UNIX based toolsets.	
Options must be preceded by '/' for non-UNIX based toolsets.	
Spaces between the options and the case of letters in parameters are not significant.	
Options may be specified in any order.	

For example, the following command would create a Makefile to produce a loadable program for a transputer network:

```
imakef network.btl
```

In this example a Makefile is generated for the object file **network.btl**, which will be created from the configuration description file **network.pgm**. The Makefile is given the name **network** (no extension is added). When the MAKE program is run on the file **network**, source files that have been updated since the last compilation of the main program are recompiled, linked files are relinked, and the bootable file **network.btl** is produced incorporating all the changes.

19.3.1 Code targets for imakef

The type of file generated when MAKE is run on a Makefile depends on the file extension specified to **imakef** when the file is created. Target code files for which **imakef** generates Makefiles are given in the following table.

Extension	Code produced
.btl	Bootable code for multitransputer programs to be run on boot from link boards.
.btr	Loadable code without bootstrap information. For dynamic loading and EPROM programming.
.b*xx*	Bootable code for single transputer programs.
.r*xx*	Non-bootable single transputer code.
.c*xx*	Linked code.
.t*xx*	Compiled code.
.lib	Library code.

The Makefile generator can also be used to create library usage files. If the file specified to **imakef** is a **.liu** file, a library usage file is created for the corresponding **.lib** file.

19.4 Format of Makefiles

The general content of Makefiles has already been described. This section describes in detail the structure of the Makefile generated by **imakef**.

A Makefile consists of two main parts:

- A list of macro definitions for commands and command options.
- A list of file dependencies in the form of *Rules*.

19.4.1 Macro definitions

Macros represent the commands that invoke the tools on the system and sets of command line options.

The following macros are inserted into all Makefiles generated by **imakef**:

```
LIBRARIAN=ilibr
OCCAM=occam
LINK=ilink
CONFIG=iconf
ADDBOOT=iboot
LIBOPT=
OCCOPT=
LINKOPT=
CONFOPT=
BOOTOPT=
```

Macros are provided in order that commands and sets of options can be changed to suit the names used at a particular instalation. Existing macros can be modified by editing the Makefile. New macros can also be added to the list.

19.4.2 Rules

Rules define the dependencies of source files on other files, and specify action strings that invoke commands to build object files. For example, the following rule defines the target program **config.btl** as being dependent on the configuration source file **config.pgm**, the separately compiled file **sc1.c4h**, and the file of declarations **header.inc**.

```
config.btl: config.pgm sc1.c4h header.inc
  $(CONFIG) config -r $(CONFOPT)
```

The first rule in a Makefile is for the main target. Succeeding rules define subcomponents of the main target, and are listed in a hierarchical manner.

Action Strings

Action strings represent the complete command line needed to rereate a specific target file. The format is similar for all tools and consists of a call to the tool via a command macro, a fixed set of parameters, a list of command line options represented by a macro, and the name of an output file. The output file is specified in order that the rebuilt file is always written to the directory that contains the source.

The **imakef** tool can generate calls to the compiler, the configurer, the linker, the librarian, and the bootstrap tool. For the compiler the call is always of the form:

$(OCCAM) *filename* **-t***x* **-***m* **$(OCCOPT)** -o *outputfile*

where: *filename* is the name of the file to be acted upon. The **.occ** extension is added by the compiler.

x is the processor type.

m is the error mode that the program is to be compiled for.

outputfile is the name of the output file to be written.

For programs configured for boot from link boards, calls to the configurer are of

the form:

```
$(CONFIG) filename $(CONFOPT)
```

The formats for calls to the linker, librarian, and bootstrap tools are as follows. Examples are shown for UNIX based and non-UNIX based toolsets:

```
$(LINK) -f filename $(LINKOPT) -o outputfile
$(LINK) /f filename $(LINKOPT) /o outputfile

$(LIBRARIAN) -f filename $(LIBOPT) -o outputfile
$(LIBRARIAN) /f filename $(LIBOPT) /o outputfile

$(ADDBOOT) filename $(BOOTOPT) -o outputfile
$(ADDBOOT) filename $(BOOTOPT) /o outputfile
```

19.4.3 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 non-occam code.

Adding options

The **imakef** program generates action strings which have the minimum of options for each tool. In most cases additional options are unnecessary; for compiler calls options can be added using the **#OPTION** compiler directive. To use the same set of options for all calls to a particular tool simply alter the macro that defines the set of options for that tool.

For example, if debugging data is to be disabled for all occam compilations then OCCOPT could be redefined to include the compiler 'D' option.

When linking a file with unusual buffer sizes it is necessary to add the required options to the existing option macro. Remember to add these options to the Makefile each time **imakef** is used.

Adding rules for C, FORTRAN and Pascal

The **imakef** program does not generate entries for C, FORTRAN and Pascal routines that are incorporated into occam programs using the **#IMPORT** directive. It does, however, generate dependency entries for language inserts within occam programs and files. To allow MAKE to incorporate code written in other languages, the Makefile can be edited by hand. Alternatively, the source

of **imakef** can be modified to work with the language system being used.

The C source of **imakef** is supplied on the toolset 'source' subdirectory. For more details see the Delivery Manual.

19.5 Library usage files

Library usage files describe the dependencies of a library on other separately compiled units and libraries. 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.

Library usage files are created using the **imakef** tool. 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.

19.6 Error Messages

Error messages produced by the **imakef** program are displayed in the standard toolset format. They may be prefixed by **Warning:** or **Error:**.

A library usage file is only valid by itself

Only a single **.liu** file can be specified on the command line.

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 make library usage file for a library with no source

The source code for a library must be present in order to create a library usage file.

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 library usage file

The library usage file cannot be opened for writing by the program.

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.

Library without a build or usage file does not exist

A library has been referenced which does not exist on the path, has no library build file (no source), and no library usage file.

Malloc failed

The program has failed while trying to dynamically allocate memory for its own use. Try using a transputer board with more memory. If the program is being run on the host it may be possible to increase the memory available using host commands.

Options are incorrectly deliminated

The terminating bracket which determines the options in a library build file, is missing at the given line number.

#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, for example, a library usage file, or a file of an unknown file type. Files with any extension can be referenced using the `#IMPORT` compiler directive.

Writing file

An host system error occurred whilst the file was being written.

20 `iserver` — host file server

This chapter describes the host file server **`iserver`**, which loads programs onto transputers and transputer networks and provides the run-time environment through which programs communicate with the host.

20.1 Introduction

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

- Loading programs and controlling transputer networks
- Runtime access to host services for application programs.

At the application program level, all communications with the host file server are through the libraries **`hostio.lib`** and **`streamio.lib`**. These are described in chapter 24.

20.2 Running the server

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

`iserver` {*options*}

where: *options* is a list of one or more options from table 20.1.

20.2.1 Supplying parameters to the program

Any text on the command line that is not a server option is passed as parameters to the program. Valid option strings will always be interpreted as server options and must not be used as program parameters.

If **`iserver`** is invoked with no options, brief help information is displayed.

20.2.2 Loading programs

Before a program can be loaded onto a transputer network it must be compiled, linked and made bootable using either the bootstrap tool **`iboot`** (for single transputer programs), or the configurer **`iconf`** (for multitransputer programs).

Option	Description
SA	Analyses the root transputer and peeks 8K of its memory.
SB *filename*	Boots the program contained in the named file.
SC *filename*	Copies the named 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 link address or device name.
SR	Resets the root transputer and subsystem on the link.
SS	Serves the link, that is, provides host system support to programs communicating on the host link.
Options must be preceded by '-' for UNIX based toolsets. Options must be preceded by '/' for non-UNIX based toolsets. Spaces between options and the case of letters in the parameters are not significant. Options may be in any order. Option SB *filename* is equivalent to SR SS SI SC *filename*.	

Table 20.1 Host file server options

The file to be loaded will have a `.btl` or a `.bxx` file extension.

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

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

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

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.

20.2.3 Terminating the server

To terminate the server press the host system break key. When the key is pressed the following prompt is displayed:

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

To terminate the server type '**x**' or press RETURN.

To suspend the server and resume the program later, type '**s**'. On DOS-based systems this option may require a host environment variable. For further information see the Delivery Manual that accompanies the release.

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

20.2.4 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 in decimal format, 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. This variable is itself overridden by the address or name specified by the '**SL**' option.

20.2.5 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. This option should only be used for programs written entirely in occam and compiled in HALT system mode. If the program is not written entirely in occam then the error flag may be set even though no error has occurred.

20.3 Server functions

This section describes the basic set of server functions. All versions of the **iserver** will support these functions, enabling programs to be used with any version of the toolset.

These functions are not intended for application programmers. They are briefly described here for those who wish to implement a server on a new host, or to add new facilities to the existing server.

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 below. Formal definitions can be found in appendix F.

File system commands

Command	Description
Fopen	Opens a file, and returns a stream identifier.
Fclose	Closes a file.
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.
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.

20.4 Error messages

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.

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.

Failed to analyse root transputer

The link driver could not analyse the transputer.

Failed to reset root transputer

The link driver could not reset the transputer.

Link name is too long, maximum size is *number* **characters**

The specified name was too long. *number* is the maximum length.

Protocol error, *message*

Incorrect protocol on the link. This can happen if there is a hardware fault, or if an incorrect version of the server is used.

message can be any of the following:

got *number* **bytes at start of a transaction**
packet size is too large
read nonsense from the link
timed out getting a further *dataname*
timed out sending reply message

For more information about server protocols see appendix F.

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 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.

21 isim — T414 simulator

This chapter describes the T414 simulator tool **isim** that simulates the operation of the T414 transputer and allows programs to be tested and debugged without recourse to hardware. The chapter explains how to invoke the tool and describes the Monitor page and symbolic debugging facilities. The chapter ends with a list of error messages.

21.1 Introduction

The simulator can run any program that can be run on a single IMS T414, on a boot from link transputer evaluation board. It provides most of the symbolic and low level debugging features of the toolset debugger, plus the abilities to set break points at source level and to single step the program.

To use the interactive facilities of the simulator the program must be compiled with debug data enabled, and the program must be linked and made bootable.

An important feature of the simulator is that the simulated code is identical to that which would run on a real transputer. Because all transputers are compatible at the occam source level, programs that run correctly using the simulator will run correctly on any real 32-bit transputer after recompiling for the correct transputer type.

The simulator can also be used to familiarise new users with the toolset and transputer programming.

21.2 Running the simulator

To invoke the simulator, use the following command line:

isim *filename programparameters*

where: *filename* is the bootable file containing the program to be simulated;

programparameters are the parameters to be passed to the program.

If **isim** is run with no filename or parameters brief help information is displayed.

When first invoked the simulator enters the Monitor page environment, which displays low level data about the simulated transputer and a list of options. Typing

'?' at the Monitor page displays a summary of the commands available.

21.2.1 The ITERM file

Like the debugger, the simulator reads the ITERM file to determine how to control the terminal screen and keyboard. The name of the ITERM file should be defined in the host environment variable **ITERM**. For details of the ITERM file see appendix D.

21.2.2 Loading and running a program

To load a program type 'X' at the Monitor page. Once loaded, the occam source can be examined by typing 'O' to enter the symbolic interface.

To run a loaded program type 'G' at the Monitor page. The program then runs until the program completes successfully, a run-time error occurs, or a break point is reached. If an error occurs the processor halts, the error flag is set, and the program can be debugged using the same symbolic and assembly level facilities as the debugger. For details of the facilities see chapter 14.

21.3 Simulator interfaces

The simulator has two command interfaces. The first is the Monitor page for low level debugging, which is entered when the simulator is invoked. The second is the symbolic interface for examining source code. Both interfaces provide the same standard debugging features as the debugger itself, and both provide additional facilities for monitoring program execution.

21.3.1 Numerical parameters

Some simulator commands require numerical parameters, such as addresses. These can be specified as simple expressions in hexadecimal format. Expressions can be the sum of two expressions, the result of subtracting one expression from another, or constants. Constants can be: **areg**, **breg**, **creg**, **iptr**, **wptr**, hexadecimal constants, or abbreviated hexadecimal constants.

Hexadecimal numbers can also be specified in an abbreviated form by prefixing the number with '%', which assumes a hexadecimal prefix of '8000....'. For example, '8000F8A' can be specified as '%F8A'.

21.4 The Monitor page

When the simulator is first invoked from the operating system, it displays the simulator Monitor page. If the program is compiled with debugging data disabled, only the Monitor page facilities can be used.

The simulator Monitor page is similar to that of the debugger, which is described in section 14.5. Data displayed at the Monitor page includes the following items:

`Iptr`	Contents of instruction pointer (address of the *next* instruction to be executed)
`Wdesc`	Contents of workspace descriptor
`Error`	Whether the error flag was set
`Halt On Error`	Whether the halt on error flag was set
`Fptr1`	Pointer to the front of the low priority active process queue
`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
`TPtr0`	Pointer to the high priority timer queue

If `Wdesc` contains the most negative address value, it will be described as 'invalid'. This normally means that no process is executing in the simulator (for example, the program may have become deadlocked). If `Wdesc` contains the address of '`Memstart`' it is displayed as such.

An asterisk displayed next to the `Iptr` or `Wdesc` pointers indicates invalid object code. To investigate the cause, use the '`M`' command to display a memory map for the program. Invalid pointers may be generated when processes become deadlocked.

The Monitor page also displays the last instruction executed, a summary of Monitor page commands, and if an error has occurred, the cause of the error. When the simulator is first invoked the Monitor page display includes a memory map of the program.

21.4.1 Monitor page commands

Many of the simulator Monitor page commands are the same as those of the debugger; for full descriptions of the commands and the functions they provide see section 14.5.

Table 21.1 summarises the simulator Monitor page commands.

A – ASCII

Displays memory contents in ASCII. See debugger.

B – Breakpoints

Sets, displays, and cancels break points at specified memory locations or procedure calls. The command displays the Breakpoint Options Page:

```
Breakpoint Options Page

1) Set breakpoint at Address

2) Set breakpoint at Procedure

3) Display breakpoints

4) Cancel breakpoint at Address

5) Cancel breakpoint at Procedure

Select Option?
```

C – Load debugging data

Loads debugging data for a specific separately compiled unit. Specify the name of the compiled file.

D – Disassemble

Displays memory in transputer instructions. See debugger.

G – Go

Starts the program, or restarts the program after it has been halted (unless the error flag has been set, in which case the program can no longer be run). The program will run until it completes successfully, sets the error flag, or reaches a break point.

Key	Meaning	Description
A	ASCII	Displays a portion of memory in ASCII.
B	Break points	Breakpoint menu.
C	Load debug data	Loads debugging data for a specific module.
D	Disassemble	Displays transputer instructions at a specified area of memory.
G	Go	Runs program (or resumes execution).
H	Hex	Displays a portion of memory in hexadecimal.
I	Inspect	Displays a portion of memory in any occam type.
L	Links	Displays **Iptr** and **Wdesc** for processes waiting for input or output on a link, or for a signal on the **Event** pin.
M	Memory map	Displays a memory map of the simulated transputer.
O	occam	Changes to symbolic debugging.
P	Procedures	Displays occam procedures and addresses.
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 one 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.
X	Boot	Runs **iboot** and loads the program.
?	Help	Displays help information.
FINISH	Quit	Quits the simulator.
CURSOR UP		Scrolls the current display.
CURSOR DOWN		Scrolls the current display.
CODE INFO	Help	Displays help information.
REFRESH	Refresh	Redraws the screen.

Table 21.1 Simulator Monitor page commands

To start the program, specify a break point address after the following prompt and press RETURN:

```
(break point address)
```

The default is not to set a break point.

H – Hex

Displays memory in hexadecimal format. See debugger.

I – Inspect

Displays a portion of memory in any occam type. See debugger.

L – Links

Displays information about links. See debugger.

M – Memory map

Displays a complete memory map. See debugger.

O – occam

Changes the simulator to symbolic debugging mode.

When entering symbolic debugging from the Monitor page, addresses for the desired `Iptr` and `Wdesc` must be supplied. Specify an address for `Iptr` after the prompt:

```
Iptr (#hhhhhhhh) ?
```

To accept the default, press ENTER. The default is the value given by the current `Iptr`.

Useful addresses can be determined using the Monitor page '`P`', '`R`', '`T`', and '`L`' commands.

If the supplied `Iptr` is not within the program body, one of the following error messages is displayed and you are returned to the Monitor page after the next key press.

```
Error: Cannot locate to configuration level code
```

```
Error : Location is not in program or a library
```

If the supplied **Iptr** is within the program body, a **Wdesc** address should be specified after the following prompt:

```
Wdesc (#hhhhhhhh) ?
```

If no symbolic features other than a single 'locate' are required, a specific **Wdesc** is not required and the default can be accepted by typing ENTER. If an invalid **Wdesc** is given, most of the symbolic features will not work, or will give incorrect results.

After values for **Iptr** and **Wdesc** have been supplied, the simulator displays the source code at the required location, and the full range of symbolic commands are available.

P – Procedures

Displays the names and addresses of all functions and procedures in the program. CURSOR UP and CURSOR DOWN can be used to scroll through the list of procedures. Information provided by this command is useful for setting break points or for locating to specific locations in the source.

Q – Quit

Quits the simulator, and returns to the operating system.

R – Run queue

Displays process queues. See debugger.

T – Timer queue

Displays timer queues. See debugger.

U – Assign

Assigns a value to a register. To assign a value, specify the register by name (abbreviations are permitted), and give a value to be assigned to the register.

X – Boot

Loads the program and executes the bootstrap code. When complete, the **Iptr** will be pointing to the start of the program. The program source can then be examined, or the program run.

[?] – Help

Lists the available commands.

[CODE INFORMATION] – Help

Lists the available commands.

[REFRESH] – Refresh

Redisplays the screen.

[FINISH] – Quit

Quits the simulator, and returns to the operating system.

The [CURSOR UP] and [CURSOR DOWN] keys can be used to scroll the display.

21.5 Symbolic facilities

Provided that the program is compiled with debugging information enabled and the source is available, the simulator allows any part of the program to be accessed in a read-only manner, using symbolic debugging.

Only the modules that are to be inspected using the symbolic options need to be compiled with debugging data enabled; the remainder need not. Compiling a module with debugging enabled does not affect the code which is produced in any way; it merely controls the production of debugging information for the debugger and simulator to use. No extra bugs will be introduced (or existing bugs masked) by recompiling the program with debugging disabled.

For convenience in the following descriptions, symbolic commands are divided into two groups; symbolic debugging commands, and program execution monitoring commands.

21.5.1 Symbolic debugging commands

The simulator symbolic debugging commands are the same as those of the debugger.

Table 21.2 lists the symbolic commands. For details of the commands see section 14.4.4.

Locating and backtracing

While locating to a source line, the simulator displays the following message:

```
'Locating...'
```

If the source line is not present the simulator locates instead to the line corresponding to the call of that code, and will repeat the locate operation until it finds some source code to display. While backtracing in this way, the simulator displays the following message:

```
'Backtracing ...'
```

In certain situations the location displayed may not correspond exactly to the expected location. For example, if no valid branch of an **IF** or **CASE** has been found, the simulator locates to the statement *following* the construct. For further details see section 7.4.

Function	Operation
BACKTRACE	Locates to procedure or function call.
TOP OF FILE	Goes to first line of the file.
BOTTOM OF FILE	Goes to last line of the file.
CHANGE FILE	Displays a different source file.
CHANNEL	Locates to the process waiting on a channel.
CODE INFORMATION	Displays a summary of utility key uses.
ENTER FILE	Changes to an included file.
EXIT FILE	Changes to an enclosing file.
GET ADDRESS	Displays address of source line.
GOTO LINE	Goes to a specific line in the file.
INFO	Displays some extra information.
INSPECT	Displays the type and value of an occam symbol.
LINKS	Displays the link connections.
MONITOR	Changes to the Monitor page.
RELOCATE	Locates back to the last location line.
RETRACE	Retraces the last operation.
SEARCH	Searches for a specified string.

Table 21.2 Symbolic debugging commands

21.5.2 Execution monitoring commands

In addition to the symbolic debugging commands, the simulator provides commands for monitoring program execution interactively. Facilities available include the setting of break points and single step execution of the code.

Table 21.3 summarises the simulator execution monitoring commands.

Function	Operation
SET BREAK	Sets and removes break points.
SINGLE STEP	Single steps a source line.
WALK	Single steps a source line, preventing the process descheduling.

Table 21.3 Execution monitoring commands

SET BREAK

Toggles a break point on the current line. If the cursor is positioned on a line that generates no code, for example, one containing only declarations or comments, the break point appears at the first true instruction *after* the line. To determine the address at which a break point has been set use the Monitor page 'B' and 'O' commands.

To remove a break point **SINGLE STEP**

Single steps to the next source line. If the next source line is in another routine or process the operation may take a little time and the message '`Locating...`' will be displayed. If an error occurs as a result of the step, the process is descheduled and the next line located to may be in another process. If the next line is in a library, and the source of the library is absent, the simulator steps to the next line in the source.

WALK

The same as SINGLE STEP, but the process is not descheduled. This ensures that the next line located to is within the current process.

Note: Because processes execute normally, the WALK operation may take a little time, particularly if the line requires input or output from other processes, and those processes are not yet ready to proceed.

21.6 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.

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.

22 `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.

22.1 Introduction

The skip tool **`iskip`** prepares a network to load a program by skipping the root transputer. It does this by setting up a route-through link on the root transputer that allows data to be exchanged between the target network and the host without running program code on the root transputer. This makes the root transputer transparent to the rest of the network. The route-through link 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`**.

22.1.1 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 part of the program. The root transputer helps to load the program onto the network and subsequently acts as a data relay between the host and the target network.

   Boards that can be used to load programs in this way are the B004 PC add-in board which contains a single T414 transputer, and the B008 PC motherboard fitted with a TRAM in slot zero to act as the root transputer. Other slots on the B008 motherboard could be used to accomodate the target sub-network.

2 Programs configured for a network that normally incorporates the root transputer can be debugged without using **`idump`** to save the contents of the root transputer to disk. Programs can be loaded onto the network beyond the root transputer and the debugger can then run on the root transputer without overwriting the program. The external network must have the correct number and arrangement of processors for the program to be loaded.

This can make debugging of single transputer programs easier where a multitransputer board is available.

22.2 Running the skip tool

To invoke the **iskip** tool use the following command line:

iskip *linknumber* {*options*}

where: *linknumber* is the root transputer link to which the target network is connected.

options is a list, in any order, of one or more options from the following table.

Option	Description
E	Monitors the subsystem error flag and terminates the server when the flag is set by the program.
R	Reset subsystem. Resets the network of transputers connected to link *linknumber*. Does not reset the root transputer.
I	Displays progress information as the tool runs.
Options are preceded by '–' for UNIX based toolsets.	
Options are preceded by '/' for non-UNIX based toolsets.	
Spaces between options and the case of letters in parameters are not significant.	
Options may be specified in any order.	

22.2.1 Monitoring the error flag

The '**E**' option directs the server to monitor the halt-on-error signal and terminate when it becomes set. The program can then be debugged. If the '**E**' option is not used the program may become suspended and the server should be terminated using the host system break key.

The '**E**' option is only used when loading programs compiled in HALT mode.

Note: There is a delay of one second after **iskip** is invoked before the error flag is monitored; if the program fails within this interval the server may not terminate correctly.

22.2.2 Loading a program

Once **iskip** has been invoked to prepare the network, the program is loaded by invoking **iserver** with the '**SS**' and '**SC**' options.

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.

For an example of how to use **iskip** to load a program see section 6.5.

22.3 Error messages

This section lists error messages that can be generated by the skip tool.

Called incorrectly

Command line parsing error. Check command line syntax.

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

You failed to give the name of a bootable file on the command line.

This option must be followed by a parameter: *option*

Option requires a parameter. Check command line syntax.

Unknown option: *option*

The specified option was invalid. Check permitted options.

You must specify a link number (0 to 3)

A link number is required. Specify the link on the root transputer to which the network is connected. If you specify the link to which the host is connected an error is reported.

23 occam — occam 2 compiler

This chapter describes the occam 2 compiler **occam**. 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 chapter ends with some features of the transputer implementation of occam supported by the occam 2 compiler, an explanation of how the compiler allocates memory, and a list of error messages.

23.1 Introduction

The toolset compiler implements the occam 2 language targetting to IMS T212, T222, M212, T414, T425 and T800 transputers. For a full description and formal definition of the occam 2 language see the '*occam 2 Reference Manual*'.

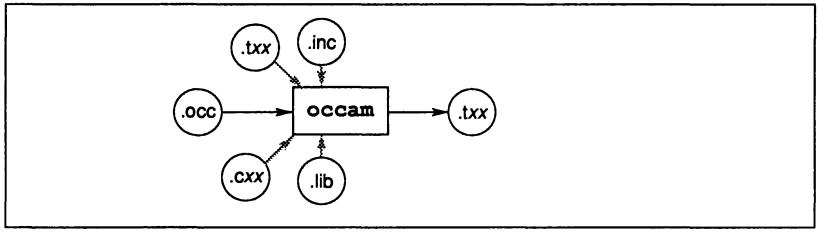
Each compilation of a program must be targetted at a specific transputer or transputer class and in one of four execution error modes. All components of a program to be run on the same transputer must be compiled for the same target processor and error mode.

Six directives, extensions to standard occam are recognised by the occam 2 compiler. These are **#USE**, **#INCLUDE**, **#IMPORT**, **#OPTION**, **#COMMENT** and **#SC**. Compiler directives are described in section 23.6.

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 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. For details of version control using Make programs and the **imakef** tool see chapter 19.

The operation of the compiler in terms of file extensions is shown below.



23.2 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:

occam *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. If the filename is omitted brief help information is displayed.

options is a list, in any order, of one or more of the options given in tables 23.1 and 23.2.

If the compilation is unsuccessful an error message is displayed giving the name of the file and the number of the line where the error occurred. Compiler error messages are listed in section 23.10.

23.2.1 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 and an extension is added in accordance with the rules described in section 3.7.2.

If you use the Makefile generator tool **imakef** to assist with version control you *must* use the recommended extensions.

23.3 Transputer targets

The compiler produces code for the IMS T212, M212, T222, T414, T425 and T800 transputers. These can be grouped into transputer classes.

23.3.1 Transputer classes

Transputers can be grouped into compilation classes because much of the object code is common between different transputers. For example, code for all the 32-bit processors (IMS T414, T425, T800) is identical provided that no floating-point CRC, or 2D block move operations are used.

32-bit processors can be grouped into three classes according to the overlap in their instruction sets. The three classes and the compatible target processors are listed below.

Class	Compatible targets
TA	T414, T425, T800
TB	T414, T425
TC	T425, T800

Code compiled for a member of one class can be mixed with code compiled for any member of the class, or for a superset of that class. For example, code compiled for class TC can be mixed with code compiled specifically for the T425 and T800 processors and with code compiled for class TA, but not with code compiled for class TB.

The formal rule for mixing code between classes is as follows:

> Code may be called provided it is compiled for a class which is the same or is a superset of the calling code.

For more details on mixing transputer types see section 4.4.2.

If you compile for TA or TC transputer classes or in UNIVERSAL error mode, you must disable the compiler libraries with the '**E**' option, because they are not supplied in these compilation modes. For details of the compiler libraries see

Option	Description
TA	Compile for transputer class TA (T414, T425, T800)
TB	Compile for transputer class TB (T414, T425)
TC	Compile for transputer class TC (T425, T800)
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.
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 T425
T800	Compile for a T800 processor.
T8	Same as T800.
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.
U	Produces code in UNDEFINED mode.
X	Produces code in UNIVERSAL mode.
A	Prevents the compiler from performing alias checking. The default is to perform alias checking. This option also disables usage checking. Details of alias and usage checking rules are given in the '*occam 2 Reference Manual*'.
C	Disables the generation of object code. The compiler performs a syntax check only.
D	Disables the generation of debugging information. The default is to produce debugging data. Debugging data is required by the debugger and by the transputer simulator.
E	Prevents the compiler from loading the compiler libraries. If this option is used and the occam code requires use of the libraries, an error is reported. The compiler libraries *must* be disabled when compiling for TA or TC transputer classes, or for any processor in UNIVERSAL error mode.

Table 23.1 occam 2 compiler options

Option	Description
G	Enables the compiler to recognise the restricted range of transputer instructions, via the GUY construct. See appendix B for the list of permitted instructions.
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.
K	Disables run-time checking of array bounds. The default is to perform run-time range checking. This option has no effect in UNDEFINED mode.
L	Loads the compiler and terminates. Useful for loading the compiler onto the board prior to compiling a program.
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*'.
O *outputfile*	Specifies the name of the output file. If no output file is specified the compiler uses the input filename and adds a file extension in accordance with the rules outlined in section 3.7.2
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.
W	Enables the compiler to recognise the full range of transputer instructions, via the GUY construct. See the *Transputer instruction set: a compiler writer's guide* for the complete list of instructions.
Options must be preceded by '–' for UNIX based toolsets Options must be preceded by '/' for non-UNIX based toolsets Spaces between options and the case of letters in parameters are not significant. Options may be in any order. N.B. some options are mutually exclusive (e.g. T8 and T2). If mutually exclusive options are given then an error is reported.	

Table 23.2 occam 2 compiler options

24.2.

23.4 Compilation error modes

The compilation mode determines the behaviour of a program if it fails during execution. There are three main modes; HALT system, STOP process, and UNDEFINED. There is also a special mode called UNIVERSAL, which is described separately in section 23.4.1.

The execution behaviour of programs compiled in the three standard modes is as follows:

HALT When an error occurs in the program the transputer halts. This is useful for developing and debugging systems and is the default 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, other processes continue until they become dependent 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 the time out is exceeded.

UNDEFINED When an error occurs the effect on code execution is not defined. UNDEFINED mode implies a minimum of run time checking, and is useful for optimising the run time of proven code.

Compilation modes and their effect are described in more detail in the '*occam 2 Reference Manual*'; also see section 4.4.3.

23.4.1 UNIVERSAL mode

It is sometimes desirable to combine processes compiled in UNDEFINED error mode, where run time checking is kept to a minimum, with code compiled in other modes.

UNIVERSAL mode is like UNDEFINED mode in that run time checking is minimised, but unlike UNDEFINED mode it allows code mixing with any of the three standard error modes. Code compiled in any of the three error modes may call code in UNIVERSAL mode, but UNIVERSAL mode may only call code in UNIVERSAL mode.

UNIVERSAL mode should not be used to allow indiscriminate mixing of code, but rather to permit fast execution of selected areas of code within other processes.

Note: Compiler library functions such as floating point and extended data types in programs cannot be used in UNIVERSAL mode, because the action of the compiler libraries differs between error modes. If you compile a program in UNIVERSAL mode you should disable the compiler libraries using the compiler '**E**' option.

23.5 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 16.

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 19.

23.6 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, FORTRAN and Pascal code, and support the insertion of comments in object code.

The directives are:

#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 compiler options to be actioned in source text
#SC	– references separately compiled units.

The **#USE** and **#SC** directives perform a similar function in occam code but **#USE** should always be used in preference. The **#SC** directive is included only for compatibility with the IMS D700D Transputer Development System (TDS).

If the compiler '**I**' option is used directives are displayed on the screen as the compilation proceeds.

23.6.1 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 rules defined in section 3.7.3.

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.

23.6.2 #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*"

where: *filename* is the name of the file to be included.

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 ten levels.

23.6.3 #USE directive

The **#USE** directive allows separately compiled occam units and libraries to be referenced from occam source. The file referenced by the #USE directive must be compiled for the same 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.

The syntax of the **#USE** directive is as follows:

#USE "*filename*"

where: *filename* is the name of the object code file. The object file can be a compiled (**.t***xx*), linked (**.c***xx*), or library (**.lib**) file. If you omit the file extension, the compiler adds an extension of the form **.t***xx*, using the rules described in section 3.7.2. The extension applied is the same as the one for the compiler output. For example, when compiling for a T800 in STOP on error mode the extension would be **.t8s**.

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.

23.6.4 #IMPORT directive

The **#IMPORT** directive allows code produced by compatible non-occam compilers to be referenced from occam programs. The code produced must also be compatible with the toolset linker **ilink**.

The syntax of the **#IMPORT** directive is as follows:

```
#IMPORT "filename"
```

where: *filename* is the name of the compiled and linked equivalent occam process. If no extension is given the *.cxx* extension is assumed. For more details about importing C, FORTRAN and Pascal processes into occam see chapter 9.

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.

For more information on how to set up calls to C, FORTRAN, and Pascal see chapter 9.

An example of how to use the **#IMPORT** directive is given below:

```
#IMPORT "cprogram.c4h"  -- Cprog()
        .
        .
        .

SEQ
  Cprog(fs, ts, flag, ws1, ws2, in, out)
  -- call C language program
```

In this program fragment one C program called **Cprog** is used. The parameters supplied in the program call, **flag**, **ws1**, **ws2**, **in**, and **out** are those of the type 3 procedural interface. See chapter 9 for more details about how to interface non-occam programs.

23.6.5 #COMMENT directive

The **#COMMENT** directive allows comments to be placed in the object code. The comment 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
```

23.6.6 #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.

N – disable usage checking.

V – disable separate vector space usage.

W – enable full code inserts.

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*}"

where: *option* 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.

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.

23.6.7 #SC directive

The **#SC** directive allows separately compiled occam units to be referenced from occam source. It works in a similar manner to **#USE**, except that libraries are not supported.

Note: **#SC** is supported for compatibility with the IMS D700D Transputer Development System (TDS). It places restrictions on the use of separately compiled code in a way that the **#USE** directive does not. For example, code referenced must be linked in a particular order. For this reason **#USE** should always be used in preference to to **#SC** when referencing separate compilation units.

#SC references are not permitted within code that is referenced by a **#USE** directive. **#USE** and **#SC** directives must not be used to reference the same code in a program.

The problems associated with linking code referenced by the **#SC** directive can be avoided by using the toolset Makefile generator **imakef** to keep track of file dependencies.

A separately compiled unit may contain other separately compiled units, which may be nested to any depth. All nested must be compiled before the enclosing unit can be compiled. Separately compiled units can reference other source files using the **#INCLUDE** directive and library files using the **#USE** directive. Separately compiled units that reference include files or libraries must be recompiled

if a common include file is changed.

The syntax of the #SC directive is as follows:

#SC "*filename*"

where: *filename* is the name of the file which contains the code of the separately compiled unit. If you omit the file extension, the compiler adds an extension of the form .t*xx*, using the rules described in section 3.7.2. The extension applied is the same as the one for the compiler output.

The first text after the #SC 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.

23.7 Implementation of usage checking

This section describes the usage checking that is implemented by the compiler.

23.7.1 Usage rules of occam 2

The usage checking rules 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.

23.7.2 Checking of non-array elements

Variables and channels which are not elements of arrays are checked according to the rules of occam 2. It is assumed that no channel is used for both input and output.

23.7.3 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. However, there are some problems with parallel replicators which are described below in section 23.7.5.

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 evaluated, 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
```

23.7.4 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
```

23.7.5 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 a single, constant subscript, a constant segment of an array (i.e. with constant base and count) or a constant segment with a single,

constant subscript. For example, the compiler reports an error in the following code because it considers the whole of the array **a** to have been used for output when **c ! 1** occurred since **a[1][2]** contains two subscripts.

```
PAR
  c IS a[1][2] :
  c ! 1
  a[0][1] ! 2
```

However, the following code is accepted because each abbreviation has just one, constant subscript.

```
PAR
  c.slice IS a[1] :
  c       IS c.slice[2] :
  c ! 1

  a[0][1] ! 2
```

Problems with replicators

The compiler has the following problems in its handling of replicators:

1 Parallel accesses to an array inside a replicator loop may be incorrectly checked against each other and flagged as errors. For example, in the following code the compiler reports the second assignment as an error even though this does not break the usage rules.

```
SEQ i = 0 FOR 10
  PAR
    a[i] := 1
    a[i + 1] := 2
```

The reason for the spurious error in this case is that the array elements which will be assigned to by the first assignment during the execution of the **SEQ** replicator will overlap those assigned to by the second assignment. To avoid this problem usage checking may be disabled in the compiler by using the compiler '**N**' option.

2 Replicated **PAR** loops are not checked properly. The compiler permits any usage of an array element within a replicated **PAR** provided the replicator index occurs within the subscript expression.

The following two programs are examples of incorrect programs accepted by the checker:

```
PAR i = 0 FOR 10
  a[i - i] := 1

PAR i = 0 FOR 4
  SEQ
    a[i] := 1
    a[i + 1] := 1
```

23.8 Memory allocation by the compiler

The code for a whole program occupies a contiguous section of memory.

23.8.1 Procedure code

The compiler places any nested procedures at lower addresses (nearer **MOSTNEG INT**) than the code for the enclosing procedure. Nested procedures are placed at increasingly higher addresses in the order in which their definitions are completed.

23.8.2 Code referenced by #SC

If a unit contains code referenced by the **#SC** directive then the code for this nested unit is loaded at a lower address. If a unit contains more than one unit referenced by the **#SC** directive then the code for the last declared unit is loaded at the lowest address.

The allocation of space for libraries and other separately compiled units referenced by the **#USE** directive is controlled by the linker. The linker can also be used to optimise specific libraries by placing them low in memory. This is achieved using the linker '**Q**' option. For more details see section 17.4.6.

23.8.3 Workspace

Workspace is allocated from higher to lower address (i.e. the workspace for a called procedure is nearer **MOSTNEG INT** than the workspace for the caller). In a **PAR** or **PRI PAR** construct the last textually defined process is allocated the lowest addressed workspace. In a replicated **PAR** construct the instance with the highest replication count is allocated the lowest workspace address.

Unless separate vector space is disabled, arrays (apart from those explicitly

placed in the workspace) are allocated in a separate data space. The allocation is done in a similar way to the allocation of workspace, except that the data space for a called procedure is at a *higher* address than the data space of its caller.

When a program is loaded onto a transputer in a network, memory is allocated in the following order starting at **MOSTNEG INT**: workspace; code; separate vector space. This allows the workspace to be given priority usage of the on-chip RAM, after the real arithmetic handling library and any libraries that are specified with the linker '**Q**' option. For more details about how memory is allocated at load time see section 11.3.3.

The variables within a single process (or procedure) are allocated so that the textually first variable is given the highest address in the current workspace.

From within a called procedure the parameters appear immediately above the local variables. When an unsized vector is declared as a formal procedure parameter an extra **VAL INT** parameter is also allocated to store the size of the array passed as the actual parameter. This size is in the units of the array. One extra parameter is supplied for each dimension of the array unsized in the call, in the order in which they appear in the declaration.

If a procedure requires separate vector space, it is supplied by the calling procedure. A pointer to the vector space supplied is given as an additional parameter after all the actual parameters of the call.

23.9 The transputer implementation of occam

This section defines the implementation of occam on the transputer supported This implementation is supported by the compiler.

23.9.1 Data representation

- The size of an **INT** (word) on an IMS T414, IMS T425, or IMS T800 transputer is 32 bits.
- The size of an **INT** (word) on an IMS T212, IMS T222, or IMS M212 transputer is 16 bits.
- Scalar variables are always allocated on a word (**INT**) boundary and occupy an integral number of words.
- **BOOL** and **BYTE** variables in arrays occupy 8 bits each. A declared array is aligned on a word boundary and occupies space rounded up to the

next word boundary. Note that an abbreviation of part of such an array might not begin on a word boundary.

- Protocol tags are represented by 8-bit values. The compiler allocates such values from 0 (**BYTE**) upwards in order of declaration.

- A **RETYPES** specification is invalid unless the alignment and size of the right-hand side is the same as for the left-hand side. Note especially that an array of **BOOL** or **BYTE** variables specified by an abbreviation (e.g. passed as a parameter) may have any alignment and so can not in general be retyped.

23.9.2 Hardware dependencies

- The number of priorities supported by the transputer is 2, so a **PRI PAR** may have two component processes. Nested **PRI PAR**s are invalid; the compiler checks this within a procedure, but does not check across procedure boundaries. A runtime check is done to compensate for this; if the program attempts a **PRI PAR** while at high priority, the error flag is set. Future releases of the compiler may check for nested **PRI PAR**s properly.

- The low priority clock increments at a rate of 15625 ticks per second, or one tick = 64 microseconds (IMS T800, IMS T414B, IMS T425, IMS T212, IMS T222, and IMS M212).

- The high priority clock increments at a rate of 1 000 000 ticks per second, or one tick = 1 microsecond (IMS T800, IMS T414B, IMS T425, IMS T212, and IMS M212).

- **TIMER** channels cannot be placed in memory with a **PLACE** statement.

23.9.3 Language and configuration

- The following directives are supported: **#INCLUDE**, **#USE**, **#COMMENT**, **#IMPORT**, **#OPTION**, and **#SC**. For more information about compiler directives see section 23.6.

- The following statements are supported: **PLACE** *name* **IN VECSPACE** and **PLACE** *name* **IN WORKSPACE**.

- The keyword **GUY** introduces a section of transputer assembly code.

- The numbers used as **PLACE** addresses are word offsets from the bottom of address space.

- A channel declared as **CHAN OF ANY** can be passed as an actual parameter in place of a formal channel parameter of *any* protocol. A channel of a specific protocol *cannot* be passed in place of a formal channel parameter of **CHAN OF ANY**. Communications on a channel declared as **CHAN OF ANY** must be identical at both ends of the channel.

- The syntax of the **PROCESSOR** statement is extended so that the keywords **T800**, **T414**, **T425**, or **T212** can be used to specify the transputer type.

23.10 Error messages

All messages produced by the compiler are in the standard toolset format. Some error messages may be preceded by the message:

```
Reading compiler library -
```

This indicates that the error occurred when the compiler was attempting to load its own library files. The compiler libraries are automatically loaded 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.

Bad format compiler option in #OPTION directive

Incorrect compiler options specified after a **#OPTION** directive.

Bad format code file

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.

Cannot find compiler libraries for this error mode

You must disable compiler libraries with the '**E**' option when compiling for TA or TC transputer classes or in UNIVERSAL error mode.

Cannot find compiler libraries for this target

You must disable compiler libraries with the **E** option when compiling for TA or TC transputer classes or in UNIVERSAL error mode.

Closing object code file

File system error. An object file could not be closed.

Closing quote missing from directive

Parameters following compiler directives must be enclosed in double quotes.

Closing source file

File system error. The source file or an include file could not be closed.

Duplicate option: *option*

Command line error. *option* was supplied more than once.

IBOARDSIZE not set up correctly

For details of how to set up the `IBOARDSIZE` host environment variable see the Delivery Manual.

Illegal compiler option in #OPTION directive

Incorrect compiler options specified after an `#OPTION` directive.

Illegal file name in directive

An invalid filename was given within the quote marks after a compiler directive.

#INCLUDE files nested too deep

The maximum depth of nesting permitted within a single compilation unit is ten.

Libraries must be referenced by a #USE directive

A library has been referenced using a #SC directive.

Line too long

The command line must not exceed 255 characters.

More than one error mode specified

The compilation error mode was specified more than once on the command line.

More than one transputer type specified

The transputer type was specified more than once on the command line.

No file name given

Command line error. A file name is required.

No file name supplied

Command line error. A file name is required.

No file name supplied with directive

No file name was specified within the quote marks.

No options supplied

At least one option is required.

Nothing in directive quotes

No parameter was found within the quote marks after the compiler directive.

Nothing in source file

The source file is empty.

Object code compiled for incompatible error mode

Code in a compilation unit has been compiled in an error mode that is incompatible with the main compilation.

Object code compiled for incompatible target

Code in a compilation unit has been compiled for a transputer target that is incompatible with the main compilation.

Object code not compatible with current version

The object code (a library or separately compiled procedure) was compiled with a incompatible compiler.

Opening object code file

File system error. The object code file could not be opened.

Opening #INCLUDE file

File system error. A file referenced by an `#INCLUDE` directive could not be opened.

Opening object code file

File system error. The object file could not be opened.

Opening quote missing from directive

Parameters following compiler directives must be enclosed in double quotes.

Opening source file

File system error. The source file or an include file could not be opened.

#OPTION directive must be at start of file

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.

Reading object code file

File system error. An object file could not be read.

Reading source file

File system error. The source file or an include file could not be read.

This option must be followed by a parameter: *option*

Command line error. *option* requires a parameter.

Unknown option: *option*

Command line error. The option *option* was invalid.

#USEd code contains #SC code

The **#USE** directive cannot be used to reference that contains code referenced by a **#SC** directive.

Writing object code file

File system error. The object code file could not be closed.

24 occam libraries

24.1 Introduction

A comprehensive set of occam libraries is provided for use with the toolset. They include the compiler libraries which are automatically referenced by the compiler, and a number of user libraries to assist with common programming tasks.

The user libraries provide standard mathematical functions, host i/o and file management procedures, file stream i/o support, and many other functions. A full list of all the toolset libraries with brief descriptions of their contents can be found in table 24.1.

24.1.1 Using the occam libraries

If a library routine is used in a program then the library must be declared with the **#USE** directive and the declaration must be in scope where the routine is used. The scope of a library, as with all occam declarations, is determined by

Library	Description
Compiler Libraries	Multiple length integer arithmetic Floating point functions 32 bit IEEE arithmetic functions 64 bit IEEE arithmetic functions 2D block move library Bit manipulation and CRC library Arithmetic instruction library
`snglmath.lib`	Single length mathematical functions
`dblmath.lib`	Double length mathematical functions
`tbmaths.lib`	T414/425 optimised maths functions
`hostio.lib`	Host file server library
`streamio.lib`	Stream i/o library
`string.lib`	String library
`convert.lib`	Type conversion library
`crc.lib`	Block CRC library
`xlink.lib`	Extraordinary link handling library
`process.lib`	Process support library

Table 24.1 Toolset libraries

its level of indentation in the occam text.

An example showing how to declare a library is given below.

```
#USE "hostio.lib"
```

Linking libraries

All libraries used by a program or program module must also be linked with the main progam. This includes the compiler libraries even though they are automatically referenced when the program is compiled.

24.1.2 Listing library contents

You can use the `ilist` tool to examine the contents of a library and determine which routines are available. The tool displays procedural interfaces for routines in each library module and the code size and workspace requirements for individual modules. It can also be used to determine the transputer types and error modes for which the code was compiled.

24.1.3 Toolset constants

Constants and protocols used by the toolset libraries are defined in four include files which are supplied with the toolset. Two of the four files provide constants and definitions for the hostio and streamio libraries, a third provides mathematical and trigonometric constants and the fourth contains the absolute transputer link addresses. All files containing constant definitions must be declared in the program before the library that references them.

Files of constants provided with the toolset are summarised in table 24.1.3. Full listings of the files can be found in appendix C.

File	Description
`hostio.inc`	Constants for the host file server interface
`streamio.inc`	Constants for the stream i/o interfaces
`mathvals.inc`	Maths constants
`linkaddr.inc`	Addresses of transputer links

Table 24.2 Files of constants

24.2 Compiler libraries

Compiler libraries are used internally by the compiler to implement multiple length and floating point arithmetic, IEEE functions, and special transputer functions such as bit manipulation and 2D block data moves. They are found automatically by the compiler on the path specified by the **ISEARCH** host environment variable and do not need to be referenced by the **#USE** directive.

The compiler libraries can be disabled using the compiler '**E**' option. The libraries *must* be disabled for programs compiled in UNIVERSAL error mode, and for programs compiled for transputer classes TA and TC.

Separate libraries are supplied for the T2, T4, and T8 families of transputers, in each main error mode. A full list of the compiler libraries is given below.

File	Processor/error mode
occam2h.lib	T212/222/M212 halt
occam2s.lib	T212/222/M212 stop
occam2u.lib	T212/222/M212 undefined
occambh.lib	T414/425 halt
occambs.lib	T414/425 stop
occambu.lib	T414/425 undefined
occam8h.lib	T800 halt
occam8s.lib	T800 stop
occam8u.lib	T800 undefined

File names of the compiler libraries must not be changed. The compiler assumes these filenames, and generates an error if they are not found on the path specified by the host environment variable **ISEARCH**.

Descriptions of some of the compiler library functions and procedures can be found in the '*occam 2 Reference Manual*'.

24.2.1 User functions

The following routines from the compiler libraries may be of interest to the applications programmer. Calls to these routines can be made directly and do not have to reference the library in a **#USE** statement.

The functions are grouped as follows: maths functions, including trigonometric and extended arithmetic routines; 2-D block moves; bit manipulation; and functions for cyclic redundancy checking (CRC).

Maths functions

Result(s)	Function	Parameter specifiers
REAL32	ABS	VAL REAL32 x
REAL32	SQRT	VAL REAL32 x
REAL32	LOGB	VAL REAL32 x
REAL32	FLOATING.UNPACK	VAL REAL32 x
REAL32	MINUSX	VAL REAL32 x
REAL32	MULBY2	VAL REAL32 x
REAL32	DIVBY2	VAL REAL32 x
REAL32	FPINT	VAL REAL32 x
BOOL	ISNAN	VAL REAL32 x
BOOL	NOTFINITE	VAL REAL32 x
REAL32	SCALEB	VAL REAL32 x, VAL INT n
REAL32	COPYSIGN	VAL REAL32 x, y
REAL32	NEXTAFTER	VAL REAL32 x, y
BOOL	ORDERED	VAL REAL32 x, y
BOOL, INT32, REAL32	ARGUMENT.REDUCE	VAL REAL32 x, y, y.err
REAL32	REAL32OP	VAL REAL32 x, VAL INT op, VAL REAL32 y
REAL32	REAL32REM	VAL REAL32 x, REAL32 y
BOOL, REAL32	IEEE32OP	VAL REAL32 x, VAL INT rm, op, VAL REAL32 y
BOOL	REAL32EQ	VAL REAL32 x, y
BOOL	REAL32GT	VAL REAL32 x, y
INT	IEEECOMPARE	VAL REAL32 x, y

Result(s)	Function	Parameter specifiers
REAL64	DSQRT	VAL REAL64 x
REAL64	DABS	VAL REAL64 x
REAL64	DLOGB	VAL REAL64 x
REAL64	DFLOATING.UNPACK	VAL REAL64 x
REAL64	DMINUSX	VAL REAL64 x
REAL64	DMULBY2	VAL REAL64 x
REAL64	DDIVBY2	VAL REAL64 x
REAL64	DFPINT	VAL REAL64 x
BOOL	DISNAN	VAL REAL64 x
BOOL	DNOTFINITE	VAL REAL64 x
REAL64	DSCALEB	VAL REAL64 x, VAL INT n
REAL64	DCOPYSIGN	VAL REAL64 x, y
REAL64	DNEXTAFTER	VAL REAL64 x, y
BOOL	DORDERED	VAL REAL64 x, y
BOOL, INT32, REAL64	DARGUMENT.REDUCE	VAL REAL64 x, y, y.err
REAL64	REAL64OP	VAL REAL64 x, VAL INT op, VAL REAL64 y
REAL64	REAL64REM	VAL REAL64 x, y
BOOL, REAL64	IEEE64OP	VAL REAL64 x, VAL INT rm, op, VAL REAL64 y
BOOL	REAL64EQ	VAL REAL64 x, y
BOOL	REAL64GT	VAL REAL64 x, y
INT	DIEEECOMPARE	VAL REAL64 x, y

Result(s)	Function	Parameter specifiers
INT	LONGADD	VAL INT left, right, carry.in
INT	LONGSUM	VAL INT left, right, carry.in
INT	LONGSUB	VAL INT left, right, borrow.in
INT, INT	LONGDIFF	VAL INT left, right, borrow.in
INT, INT	LONGPROD	VAL INT left, right, carry.in
INT, INT	LONGDIV	VAL INT dividend.hi, dividend.lo, divisor
INT, INT	SHIFTRIGHT	VAL INT hi.in, lo.in, places
INT, INT	SHIFTLEFT	VAL INT hi.in, lo.in, places
INT, INT, INT	NORMALISE	VAL INT hi.in, lo.in
INT	ASHIFTRIGHT	VAL INT operand, places
INT	ASHIFTLEFT	VAL INT argument, places
INT	ROTATERIGHT	VAL INT argument, places

2D block moves

Procedure	Parameter Specifiers
MOVE2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length
DRAW2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length
CLIP2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length

Procedure definitions

MOVE2D

```
PROC MOVE2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

Moves a data block of size `width` by `length` starting at byte `Source[sy][sx]` to the block starting at `Dest[dy][dx]`.

DRAW2D

```
PROC DRAW2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

As MOVE2D but only non-zero bytes are transferred.

CLIP2D

```
PROC CLIP2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

As MOVE2D but only zero bytes are transferred.

Bit manipulation functions

Result	Function	Parameter Specifiers
INT	BITCOUNT	VAL INT Word, CountIn
INT	BITREVNBITS	VAL INT x, n
INT	BITREVWORD	VAL INT x

Function definitions

BITCOUNT

```
INT FUNCTION BITCOUNT (VAL INT Word, CountIn)
```

Counts the number of bits set to 1 in `Word`, adds it to `CountIn`, and returns the total.

BITREVNBITS

```
INT FUNCTION BITREVNBITS (VAL INT x, n)
```

Returns an **INT** containing the **n** least significant bits of **x**.

BITREVWORD

```
INT FUNCTION BITREVWORD (VAL INT x)
```

Returns an **INT** containing the bit reversal of **x**.

CRC functions

Result	Function	Parameter Specifiers
INT	CRCWORD	VAL INT data, CRCIn, generator
INT	CRCBYTE	VAL INT data, CRCIn, generator

Function descriptions

CRCWORD

```
INT FUNCTION CRCWORD  (VAL INT data, CRCIn,
                       generator)
```

Performs a cyclic redundancy check over the single word **data** using the CRC value obtained from the previous call. **generator** is the CRC polynomial generator. Can be used iteratively on a sequence of words to obtain the CRC.

CRCBYTE

```
INT FUNCTION CRCBYTE (VAL INT data, CRCIn,
                      generator)
```

As **CRCWORD** but performs the check over a single byte. The byte processed is contained in the least significant end of the word **data**.

For further infomation about CRC functions see '*INMOS Technical note 26: Notes on graphics support and performance improvements on the IMS T800*'.

24.3 Maths libraries

Elementary maths and trigonometric functions are provided in three libraries, as follows:

Library	Description
`snglmath.lib`	Single length library
`dblmath.lib`	Double length library
`tbmaths.lib`	T414/T425 optimised library

The single and double length libraries contain the same set of functions in single and double length forms. By convention the double length forms begin with the letter 'D'. Function names are in upper case.

The T414/T425 library is a combined single and double length library containing all the single and double length functions optimised for the T414 and T425 processors. The standard maths libraries can also be used on the T414 and T425, but optimum performance on these processors can be achieved by using the optimised functions.

The accuracy of the T414/T425 optimised functions is similar to that of the standard single length functions but results returned may not be identical because different algorithms are used.

Functions in the optimised librray have the same names as the equivalent functions in the single and double length libraries. This means that the optimised library cannot be used together with either the single or double length library on the same processor. If the optimised library is used in code compiled for any processor except a T414 or T425, the compiler reports an error.

Single and double length maths functions are listed below. Descriptions of the functions can be found in succeeding sections.

Result(s)	Function	Parameter specifiers
REAL32	ALOG	VAL REAL32 X
REAL32	ALOG10	VAL REAL32 X
REAL32	EXP	VAL REAL32 X
REAL32	POWER	VAL REAL32 X, VAL REAL32 Y
REAL32	SIN	VAL REAL32 X
REAL32	COS	VAL REAL32 X
REAL32	TAN	VAL REAL32 X
REAL32	ASIN	VAL REAL32 X
REAL32	ACOS	VAL REAL32 X
REAL32	ATAN	VAL REAL32 X
REAL32	ATAN2	VAL REAL32 X, VAL REAL32 Y
REAL32	SINH	VAL REAL32 X
REAL32	COSH	VAL REAL32 X
REAL32	TANH	VAL REAL32 X
REAL32, INT32	RAN	VAL INT32 X

Result(s)	Function	Parameter specifiers
REAL64	DALOG	VAL REAL64 X
REAL64	DALOG10	VAL REAL64 X
REAL64	DEXP	VAL REAL64 X
REAL64	DPOWER	VAL REAL64 X, VAL REAL64 Y
REAL64	DSIN	VAL REAL64 X
REAL64	DCOS	VAL REAL64 X
REAL64	DTAN	VAL REAL64 X
REAL64	DASIN	VAL REAL64 X
REAL64	DACOS	VAL REAL64 X
REAL64	DATAN	VAL REAL64 X
REAL64	DATAN2	VAL REAL64 X, VAL REAL64 Y
REAL64	DSINH	VAL REAL64 X
REAL64	DCOSH	VAL REAL64 X
REAL64	DTANH	VAL REAL64 X
REAL64, INT64	DRAN	VAL INT64 X

24.3.1 Function definitions

```
ALOG
DALOG

      REAL 32 FUNCTION ALOG (VAL REAL32 X)
      REAL 64 FUNCTION DALOG (VAL REAL64 X)
```

Compute $\log_e(X)$.

```
ALOG10
DALOG10

      REAL32 FUNCTION ALOG10 (VAL REAL32 X)
      REAL64 FUNCTION DALOG10 (VAL REAL64 X)
```

Compute $\log_{10}(X)$.

```
EXP
DEXP

      REAL32 FUNCTION EXP (VAL REAL32 X)
      REAL64 FUNCTION DEXP (VAL REAL64 X)
```

Compute e^X.

```
POWER
DPOWER

      REAL32 FUNCTION POWER (VAL REAL32 X,
                             VAL REAL32 Y)
      REAL64 FUNCTION DPOWER (VAL REAL64 X,
                              VAL REAL64 Y)
```

Compute X^Y.

```
SIN
DSIN

      REAL32 FUNCTION SIN (VAL REAL32 X)
      REAL64 FUNCTION DSIN (VAL REAL64 X)
```

Compute sine(X) (where X is in radians).

COS
DCOS

```
REAL32 FUNCTION COS (VAL REAL32 X)
REAL64 FUNCTION DCOS (VAL REAL64 X)
```

Compute cosine(X) (where X is in radians).

TAN
DTAN

```
REAL32 FUNCTION TAN (VAL REAL32 X)
REAL64 FUNCTION DTAN (VAL REAL64 X)
```

Compute tan(X) (where X is in radians).

ASIN
DASIN

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
REAL64 FUNCTION DASIN (VAL REAL64 X)
```

Compute $\sin^{-1}(X)$ (in radians).

ACOS
DACOS

```
REAL32 FUNCTION ACOS (VAL REAL32 X)
REAL64 FUNCTION DACOS (VAL REAL64 X)
```

Compute $\cos^{-1}(X)$ (in radians).

ATAN
DATAN

```
REAL32 FUNCTION ATAN (VAL REAL32 X)
REAL64 FUNCTION DATAN (VAL REAL64 X)
```

Compute $\tan^{-1}(X)$ (in radians).

ATAN2
DATAN2

```
REAL32 FUNCTION ATAN2 (VAL REAL32 X,
                       VAL REAL32 Y)
REAL64 FUNCTION DATAN2 (VAL REAL64 X,
                        VAL REAL64 Y)
```

Compute the angular co-ordinate $\tan^{-1}(Y/X)$ (in radians) of a point whose X and Y co-ordinates are given.

SINH
DSINH

```
REAL32 FUNCTION SINH (VAL REAL32 X)
REAL64 FUNCTION DSINH (VAL REAL64 X)
```

Compute $\sinh(X)$.

COSH
DCOSH

```
REAL32 FUNCTION COSH (VAL REAL32 X)
REAL64 FUNCTION DCOSH (VAL REAL64 X)
```

Compute $\cosh(X)$.

TANH
DTANH

```
REAL32 FUNCTION TANH (VAL REAL32 X)
REAL64 FUNCTION DTANH (VAL REAL64 X)
```

Compute $\tanh(X)$.

RAN
DRAN

```
REAL32,INT32 FUNCTION RAN (VAL INT32 X)
REAL64,INT64 FUNCTION DRAN (VAL INT64 X)
```

These produce a pseudo-random sequence of integers, or a corresponding sequence of floating-point numbers between zero and one. **X** is the seed integer that initiates the sequence.

24.4 Host file server library

The host file server library contains routines that are used to communicate with the host file server. The routines are independent of the host on which the server is running. Using routines from this library you can guarantee that programs will be portable across all implementations of the toolset.

The library is compiled for T212 in universal error mode, and TA in universal error mode. Constant and protocol definitions for the hostio library, including error and return codes, is provided in the include file **hostio.inc**. A listing of the file can be found in appendix C.

24.4.1 Errors and the C run time library

The hostio routines use functions provided by the host file server. These are defined in appendix F. The server functions in turn use routines in a C run time library, some of which is implementation dependent.

In particular, the hostio routines do not check the validity of stream identifiers, and the consequences of specifying an incorrect **streamid** may differ from system to system. For example, some systems may return an error tag, some may return a text message. If you use only those stream ids returned by the hostio routines that open files (**so.open**, **so.open.temp**, and **so.popen.read**), invalid ids are unlikely to occur.

It is also possible in rare circumstances for a program to fail altogether with an invalid streamid because of the way the C library is implemented on the system. This error can only occur if direct use of the library to perform the operation would produce the same error.

24.4.2 Inputting real numbers

Routines for inputting real numbers only accept numbers in the standard occam format for **REAL** numbers. Programs that allow other ways of specifying real numbers must convert to the occam format before presenting them to the library procedure.

For details of occam syntax for real numbers see the '*occam 2 Reference Manual*'.

24.4.3 Procedure descriptions

In the procedure descriptions, **fs** is the channel *from* the host file server, and **ts** is the channel *to* the host file server. The **SP** protocol used by the host file server channels is defined in the include file **hostio.inc**, which is listed in appendix C.

The hostio routines are divided into six groups: five groups that reflect function and use, and a sixth miscellaneous group. The five specific groups are:

- File access and management
- General host access
- Keyboard input
- Screen output
- File output.

Each group of routines is described in a separate section. Each section begins with a list of the routines in the group with their formal parameters. This is followed by a formal description of each routine in turn.

24.4.4 File access routines

This group includes routines for managing file streams, for opening and closing files, and for reading and writing blocks of data.

Procedure	Parameter Specifiers
so.open	CHAN OF SP fs, ts, VAL []BYTE name, VAL BYTE type, mode, INT32 streamid, BYTE result
so.open.temp	CHAN OF SP fs, ts, VAL BYTE type, [so.temp.filename.length]BYTE filename, INT32 streamid, BYTE result
so.popen.read	CHAN OF SP fs, ts, VAL []BYTE filename, VAL []BYTE path.variable.name, VAL BYTE open.type, INT full.len, []BYTE full.name, INT32 streamid, BYTE result
so.close	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.read	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data
so.write	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, INT length
so.gets	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data, BYTE result
so.puts	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, BYTE result
so.flush	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result

Procedure	Parameter Specifiers
so.seek	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT32 offset, origin, BYTE result
so.tell	CHAN OF SP fs, ts, VAL INT32 streamid, INT32 position, BYTE result
so.test.exists	CHAN OF SP fs, ts, VAL []BYTE filename, BOOL exists
so.eof	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.ferror	CHAN OF SP fs, ts, VAL INT32 streamid, INT32 error.no, INT length, []BYTE message, BYTE result
so.remove	CHAN OF SP fs, ts, VAL []BYTE name, BYTE result
so.rename	CHAN OF SP fs, ts, VAL []BYTE oldname, newname, BYTE result

Procedure definitions

so.open

```
PROC so.open  (CHAN OF SP fs, ts,
               VAL []BYTE name,
               VAL BYTE type, mode,
               INT32 streamid, BYTE result)
```

Opens the file given by **name** and returns a stream identifier **streamid** for all future operations on the file until it is closed. File type is specified by **type** and the mode of opening by **mode**.

type can take the following values:

spt.binary	File contains raw bytes only.
spt.text	File contains text records separated by newline sequences.

mode can take the following values:

spm.input	Open existing file for reading.
spm.output	Open new file, or truncate an existing one, for writing.
spm.append	Open a new file, or append to an existing one, for writing.
spm.existing.update	Open and existing file for update (reading and writing), starting at beginning of the file.
spm.new.update	Open new file, or truncate existing one, for update.
spm.append.update	Open new file, or append to an existing one, for update.

result can take the following values:

spr.ok	The open was successful.
spr.operation.failed	The open failed.
spr.bad.name	Invalid file name.
spr.bad.type	Invalid file type.
spr.bad.mode	Invalid open mode.
spr.bad.packet.size	File name too large.

so.open.temp

```
PROC so.open.temp
        (CHAN OF SP fs, ts,
         VAL BYTE type,
         [so.temp.filename.length]BYTE filename,
         INT32 streamid, BYTE result)
```

Opens a temporary file in **spm.update** mode. If the file already exists the **nn** suffix on the name **temp***nn* is incremented up to a maximum of 9999 until an unused number is found. If the number exceeds 2 digits the last character of **temp** is overwritten. For example: if the number exceeds 99 the **p** is overwritten , as in **tem999**; if the number exceeds 999, the **m** is overwritten, as in **te9999**. File type can be **spt.binary** or **spt.text**, as with **so.open**. The name of the file actually opened is returned in **filename**.

The result returned can take any of the following values:

spr.ok The open was successful.

so.popen.read

```
PROC so.popen.read
                (CHAN OF SP fs, ts,
                 VAL []BYTE filename,
                 VAL []BYTE path.variable.name,
                 VAL BYTE open.type,
                 INT full.len, []BYTE full.name,
                 INT32 streamid, BYTE result)
```

As **so.open** but the search path for the file can be specified as a string in **path.variable.name**(defined by **ISEARCH**. Obeys the path searching rules used by the toolset, as described in section 3.7.3. File type can be
spt.binary or **spt.text**, as with **so.open**. The mode of opening is always **spm.input**.

The name of the file opened is returned in **full.name**, and the length of the file name is returned in **full.len**. If no file is opened **full.len** is set to zero.

The result returned can take any of the following values:

spr.ok	The open was successful.
spr.operation.failed	The open failed.
spr.bad.name	Null name supplied.
spr.bad.type	Invalid file type specified.
spr.bad.packet.size	File name too large.

so.close

```
PROC so.close (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               BYTE result)
```

Closes the stream identified by **streamid**.

The result returned can take any of the following values:

spr.ok	The close was successful.
spr.operation.failed	The close failed.

so.read

```
PROC so.read  (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               INT length, []BYTE data)
```

Reads a block of bytes from the specified stream up to a maximum given by the size of the array **data**. If **length** returned is not the same as the size of **data** then the end of the file has been reached or an error has occurred.

so.write

```
PROC so.write (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               VAL []BYTE data,
               INT length.written)
```

Writes a block of data to the specified stream. If **length.written** is less than the size of **data** then an error has occurred.

so.gets

```
PROC so.gets (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              INT length, []BYTE data,
              BYTE result)
```

Reads a line from the specified input stream. Characters are read until a newline sequence is found, the end of the file is reached, or all characters in **data** have been read. The newline sequence is not included in the returned array. If the read fails then either the end of file has been reached or an error has occurred.

The result returned can take any of the following values:

spr.ok	The read was successful.
spr.operation.failed	The read failed.
spr.bad.packet.size	**data** is too large (> **sp.max.readbuffer.size**).
spr.buffer.overflow	The line was larger than the buffer **data**. **length** contains the size of the buffer.

so.puts

```
PROC so.puts (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              VAL []BYTE data, BYTE result)
```

Writes a line to the specified output stream. A newline sequence is added to the end of the line. The size of **data** must be less than or equal to the hostio constant **sp.max.writebuffer.size**.

The result returned can take any of the following values:

spr.ok	The write was successful.
spr.operation.failed	The write failed.
spr.bad.packet.size	**SIZE data** is too large (> **sp.max.writebuffer.size**).

so.flush

```
PROC so.flush (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               BYTE result)
```

Flushes the specified output stream. All internally buffered data is written to the stream. Write and put operations that are directed to standard output are flushed automatically.

The result returned can take any of the following values:

spr.ok	The flush was successful.
spr.operation.failed	The flush failed.

so.seek

```
PROC so.seek (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              VAL INT32 offset, origin,
              BYTE result)
```

Sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be **offset** bytes from the position defined by **origin**. For a text file **offset** must be zero or a value returned by **so.tell**, in which case **origin** must be **spo.start**.

`origin` may take the following values:

`spo.start`	The start of the file.
`spo.current`	The current position in the file.
`spo.end`	The end of the file.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.operation.failed`	The seek failed.
`spr.bad.origin`	Invalid origin.

so.tell

```
PROC so.tell (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              INT32 position, BYTE result)
```

Returns the current file position for the specified stream.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.operation.failed`	The tell failed.

so.eof

```
PROC so.eof (CHAN OF SP fs, ts,
             VAL INT32 streamid, BYTE result)
```

Tests whether the specified stream has reached the end of a file.

The result returned can take any of the following values:

`spr.ok`	End of file has been reached.
`spr.operation.failed`	The end of file has not been reached.

so.ferror

```
PROC so.ferror (CHAN OF SP fs, ts,
                VAL INT32 streamid,
                INT32 error.no, INT length,
                []BYTE message, BYTE result)
```

Indicates whether an error has occurred on the specified stream. The integer **error.no** is a host defined error number. The message will have **length** zero if no message can be provided.

The result returned can take any of the following values:

spr.ok	An error has occurred.
spr.operation.failed	No error has occurred.
spr.buffer.overflow	An error has occurred but the message is too large for the buffer.

If the buffer overflows **length** is set to the buffer size.

so.remove

```
PROC so.remove (CHAN OF SP fs, ts,
                VAL []BYTE name, BYTE result)
```

Deletes the specified file.

The result returned can take any of the following values:

spr.ok	The delete was successful.
spr.operation.failed	The delete failed.
spr.bad.name	Null name supplied.
spr.bad.packet.size	**SIZE name** is too large (> **sp.max.removename.size**).

so.rename

```
PROC so.rename (CHAN OF SP fs, ts,
                VAL []BYTE oldname, newname,
                BYTE result)
```

Renames the specified file.

The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.operation.failed	The rename failed.
spr.bad.name	Null name supplied.
spr.bad.packet.size	File names are too large (**SIZE name1** + **SIZE name2** > **sp.max.renamename.size**).

so.test.exists

```
PROC so.test.exists (CHAN OF SP fs, ts,
                     VAL []BYTE filename,
                     BOOL exists)
```

Tests if the specified file exists. The value of **exists** is TRUE if the file exists, otherwise it is FALSE.

24.4.5 General host access

This group contains routines to access the host computer for system information and services.

Procedure	Parameter Specifiers
so.commandline	CHAN OF SP fs, ts, VAL BYTE all, INT length, []BYTE string, BYTE result
so.parse.command.line	CHAN OF SP fs, ts, VAL [][]BYTE option.strings, VAL []INT option.parameters.required, []BOOL option.exists, [][2]INT option.parameters, INT error.len, []BYTE line
so.getenv	CHAN OF SP fs, ts, VAL []BYTE name, INT length, []BYTE value, BYTE result
so.time	CHAN OF SP fs, ts, INT32 localtime, UTCtime

Procedure	Parameter Specifiers
so.system	CHAN OF SP fs, ts, VAL []BYTE command, INT32 status, BYTE result
so.exit	CHAN OF SP fs, ts, VAL INT32 status
so.core	CHAN OF SP fs, ts, VAL INT32 offset, INT length, []BYTE data, BYTE result
so.version	CHAN OF SP fs, ts, BYTE version, host, os, board

Procedure definitions

so.commandline

```
PROC so.commandline (CHAN OF SP fs, ts,
                     VAL BYTE all, INT length,
                     []BYTE string, BYTE result)
```

Returns the command line passed to the server when it was invoked. If **all** has the value **sp.short.commandline** then all options are stripped from the command line. If **all** is **sp.whole.commandline** then the command line is returned exactly as it was invoked.

The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.buffer.overflow	Command line too long for **string**.

If the buffer overflows **length** is set to the buffer size.

so.parse.command.line

```
PROC so.parse.command.line
          (CHAN OF SP fs, ts,
           VAL [][]BYTE option.strings,
           VAL []INT option.parameters.required,
           []BOOL option.exists,
           [][2]INT option.parameters,
           INT error.len, []BYTE line)
```

This procedure reads the command line and parses it for specified options and associated parameters.

The parameter **option.strings** contains a list of all the possible options. Options may be any length up to 255 bytes and are *not* case sensitive. To read a parameter that has no preceding option (such as a file name) then the first option string should be empty (contain only spaces). For example, consider a program can be supplied with a file name, and any of three options 'A', 'B' and 'C'. The array **option.strings** would look like this:

```
VAL option.strings IS [" ", "A", "B", "C"]:
```

The parameter **option.parameters.required** indicates if the corresponding option (in **option.strings**) requires a parameter. The values it may take are:

spopt.never	Never takes a parameter.
spopt.maybe	Optionally takes a parameter.
spopt.always	Must take a parameter.

Continuing the above example, the file name must be supplied and none of the options take parameters, except for 'C', which may or may not have a parameter, then **option.parameters.required** would look like this:

```
VAL option.parameters.required IS
    [sopt.always, sopt.never,
     sopt.never, sopt.maybe]:
```

If an option was present on the command line **option.exists** is set to TRUE, otherwise it is set to FALSE.

If an option was followed by a parameter then the position in the array **line** where the parameter starts and the length of the parameter are given by the first and second elements respectively of the parameter element in **option.parameters**.

If an error occurs whilst the command line is being parsed then **error.len** will be greater than zero and **line** will contain an error message of the given length. If no error occurs then **line** will contain the command line as supplied by the host file server.

so.getenv

```
PROC so.getenv (CHAN OF SP fs, ts,
                VAL []BYTE name,
                INT length, []BYTE value,
                BYTE result)
```

Returns the string defined for the host environment variable **name**. If **name** is not defined on the system **result** takes the value **spr.operation.failed**.

The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.bad.name	The specified name is a null string.
spr.operation failed	Could not read environment string.
spr.bad.packet.size	**SIZE name** is too large (> **sp.max.getenvname.size**).
spr.buffer.overflow	Environment string too large for **value**.

If the buffer overflows **length** is set to the buffer size.

so.time

```
PROC so.time (CHAN OF SP fs, ts,
              INT32 localtime, UTCtime)
```

Returns the Coordinated Universal Time and local time if they are available on the system. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If either time is unavailable then it will have a value of zero.

so.system

```
PROC so.system (CHAN OF SP fs, ts,
                VAL []BYTE command,
                INT32 status, BYTE result)
```

Passes the string **command** to the host command processor for execution. If the command string is of zero length **result** takes the value **spr.ok** if there is a host command processor, otherwise an error is returned. If **command** is non-zero in length then **status** contains the host-specified value of the command, otherwise it is undefined.

The result returned can take any of the following values:

spr.ok	Host command processor exists.
spr.bad.packet.size	The array **command** is too large (> **sp.max.systemcommand.size**).

so.exit

```
PROC so.exit (CHAN OF SP fs, ts,
              VAL INT32 status)
```

Terminates the server, which returns the value of **status** to its caller. If **status** has the special value **sps.success** then the server will terminate with a host specific 'success' result. If **status** has the special value **sps.failure** then the server will terminate with a host specific 'failure' result.

so.core

```
PROC so.core (CHAN OF SP fs, ts,
              VAL INT32 offset,
              []BYTE data, BYTE result)
```

Returns the contents of the root transputer's memory as peeked from the transputer when **iserver** is invoked with the analyse ('**SA**') option. The start of the memory segment is given by **offset**, and the number of bytes by the size of the **data** vector. An error is returned if **offset** is larger than the total amount of peeked memory.

The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.operation.failed	The operation failed.
spr.bad.packet.size	The array **data** is too large (> **sp.max.corerequest.size**).

This procedure can also be used to determine whether the memory was peeked (whether the server was invoked with the '**SA**' option), by specifying a size of zero for **data** and **offset**. If the routine fails the memory was not peeked.

so.version

```
PROC so.version (CHAN OF SP fs, ts,
                 BYTE version, host, os, board)
```

Returns version information about the server and the host on which it is running. A value of zero for any of the items indicates that the information is unavailable.

The version of the server is given by **version**. The value should be divided by ten to yield the true version number. For example, a value of 15 means version 1.5.

The host machine type is given by **host**, and can take any of the following values:

sph.PC	IBM PC
sph.NECPC	NEC PC
sph.VAX	DEC VAX
sph.SUN3	SUN Microsystems Sun-3
sph.SUN4	SUN Microsystems Sun-4

Values up to 127 are reserved for use by INMOS.

The host operating system is given by **os**, and can take any of the following values:

spo.DOS	DOS
spo.HELIOS	HELIOS
spo.VMS	VMS
spo.SunOS	SunOS

Values up to 127 are reserved for use by INMOS.

The interface board type is given by **board**, and can take any of the following values:

spb.B004	IMS B004
spb.B008	IMS B008
spb.B010	IMS B010
spb.B011	IMS B011
spb.B014	IMS B014
spb.DRX11	DRX-11
spb.QT0	Caplin QT0

Values up to 127 are reserved for use by INMOS.

24.4.6 Keyboard input

Procedure	Parameter Specifiers
so.pollkey	CHAN OF SP fs, ts, BYTE key, result
so.getkey	CHAN OF SP fs, ts, BYTE key, result
so.read.line	CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result
so.read.echo.line	CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result
so.ask	CHAN OF SP fs, ts, VAL []BYTE prompt, replies, VAL BOOL display.possible.replies, VAL BOOL echo.reply, INT reply.number
so.read.echo.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.hex.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.int64	CHAN OF SP fs, ts, INT64 n, BOOL error
so.read.echo.hex.int64	CHAN OF SP fs, ts, INT64 n, BOOL error
so.read.echo.any.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.real32	CHAN OF SP fs, ts, REAL32 n, BOOL error
so.read.echo.real64	CHAN OF SP fs, ts, REAL64 n, BOOL error

Procedure definitions

so.pollkey

```
PROC so.pollkey (CHAN OF SP fs, ts,
                 BYTE key, result)
```

Reads a single character from the keyboard. If no key is available then it returns immediately with **spr.operation.failed**. The key is not echoed on the screen.

The result returned can take any of the following values:

spr.ok	The read was successful.
spr.operation.failed	The read failed.

so.getkey

```
PROC so.getkey (CHAN OF SP fs, ts,
                BYTE key, result)
```

As **so.pollkey** but waits for a key if none is available. The key is not echoed on the screen.

The results can take the same values as **so.pollkey**.

so.read.line

```
PROC so.read.line (CHAN OF SP fs, ts, INT len,
                   []BYTE line, BYTE result)
```

Reads a line of text from the keyboard, without echoing it on the screen. The line is read until 'RETURN' is pressed at the keyboard.

The result returned can take any of the following values:

spr.ok	The read was successful.
spr.operation.failed	The read failed.

so.read.echo.line

```
PROC so.read.echo.line (CHAN OF SP fs, ts,
                        INT len, []BYTE line,
                        BYTE result)
```

As **so.read.line**, but user input is echoed on the screen.

so.ask

```
PROC so.ask (CHAN OF SP fs, ts,
             VAL []BYTE prompt, replies,
             VAL BOOL display.possible.replies,
             VAL BOOL echo.reply,
             INT reply.number)
```

Prompts on the screen for a user response on the keyboard. The prompt is specified by the string **prompt**, and the list of permitted relies by the string **replies**. Only single character responses are permitted, and alphabetic characters are *not* case sensitive. For example if the permitted responses are 'Y', 'N' and 'Q' then the **replies** string would contain the characters **"YNQ"**, and 'y', 'n' and 'q' would also be accepted. **reply.number** indicates which response was typed, numbered from zero.

If **display.possible.replies** is TRUE the permitted replies are displayed on the screen. If **echo.reply** is TRUE the user's response is displayed.

The procedure will not return until a valid response has been typed.

so.read.echo.int

```
PROC so.read.echo.int (CHAN OF SP fs, ts, INT n,
                       BOOL error)
```

Reads a decimal integer typed at the keyboard and displays it on the screen. The number must be terminated by 'RETURN'. The boolean **error** is set to TRUE if an invalid integer is typed.

so.read.echo.int64

```
PROC so.read.echo.int64 (CHAN OF SP fs, ts,
                         INT64 n, BOOL error)
```

As **so.read.echo.int** but reads 64-bit numbers.

so.read.echo.hex.int

```
PROC so.read.echo.hex.int (CHAN OF SP fs, ts,
                           INT n, BOOL error)
```

As **so.read.echo.int** but reads a number in hexadecimal format. The number must be prefixed with either '#', which directly indicates a hexadecimal number, or '%', which assumes the prefix **#8000...**.

so.read.echo.hex.int64

```
PROC so.read.echo.hex.int64 (CHAN OF SP fs, ts,
                             INT64 n, BOOL error)
```

As **so.read.echo.hex.int** but reads 64-bit numbers.

so.read.echo.any.int

```
PROC so.read.echo.any.int (CHAN OF SP fs, ts,
                           INT n, BOOL error)
```

As **so.read.echo.int** but accepts numbers in either decimal or hexadecimal format. Hexadecimal numbers must be prefixed with either '#', which specifies the number directly, or '%', which assumes the prefix **#8000...**.

so.read.echo.real32

```
PROC so.read.echo.real32 (CHAN OF SP fs, ts,
                          REAL32 n, BOOL error)
```

Reads a real number typed at the keyboard and displays it on the screen. The number must be terminated by 'RETURN'. The boolean variable **error** is set to TRUE if an invalid number is typed.

so.read.echo.real64

```
PROC so.read.echo.real64 (CHAN OF SP fs, ts,
                          REAL64 n, BOOL error)
```

As **so.read.echo.real32**, but for 64-bit real numbers.

24.4.7 Screen output

Procedure	Parameter Specifiers
so.write.char	CHAN OF SP fs, ts, VAL BYTE char
so.write.nl	CHAN OF SP fs, ts
so.write.string	CHAN OF SP fs, ts, VAL []BYTE string
so.write.string.nl	CHAN OF SP fs, ts, VAL []BYTE string
so.write.int	CHAN OF SP fs, ts, VAL INT n, field
so.write.int64	CHAN OF SP fs, ts, VAL INT64 n, VAL INT field
so.write.hex.int	CHAN OF SP fs, ts, VAL INT n, width
so.write.hex.int64	CHAN OF SP fs, ts, VAL INT64 n, VAL INT width
so.write.real32	CHAN OF SP fs, ts, VAL REAL32 r, VAL INT Ip, Dp
so.write.real64	CHAN OF SP fs, ts, VAL REAL64 r, VAL INT Ip, Dp

Procedure definitions

so.write.char

```
PROC so.write.char (CHAN OF SP fs, ts,
                    VAL BYTE char)
```

Writes the single byte `char` to the screen.

so.write.nl

```
PROC so.write.nl (CHAN OF SP fs, ts)
```

Writes a newline sequence to the screen.

so.write.string

```
PROC so.write.string (CHAN OF SP fs, ts,
                      VAL []BYTE string)
```

Writes the string **string** to the screen.

so.write.string.nl

```
PROC so.write.string.nl (CHAN OF SP fs, ts,
                         VAL []BYTE string)
```

As **so.write.string**, but appends a newline sequence to the end of the string.

so.write.int

```
PROC so.write.int (CHAN OF SP fs, ts,
                   VAL INT n, field)
```

Writes the value **n** (of type **INT**) to the screen as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified **field** width. If the **field** width is too small for the number it is widened as necessary; a zero value for **field** specifies minimum width.

so.write.int64

```
PROC so.write.int64 (CHAN OF SP fs, ts,
                     VAL INT64 n, VAL INT field)
```

As **so.write.int** but for 64-bit integers. The **field** parameter behaves as in **so.write.int**.

so.write.hex.int

```
PROC so.write.hex.int (CHAN OF SP fs, ts,
                       VAL INT n, width)
```

Writes the value **n** (of type **INT** to the screen as hexadecimal ASCII digits, preceded by the '#' character. The number of characters to be printed is specified by **width**. If **width** is larger than the size of the number then the number is padded with leading '0's or 'F's as appropriate. If **width** is smaller than the size of the number, the number is truncated to **width** digits.

so.write.hex.int64

```
PROC so.write.hex.int64 (CHAN OF SP fs, ts,
                         VAL INT64 n,
                         VAL INT width)
```

As **so.write.hex.int** but for 64-bit integers. The **width** parameter behaves as in **so.write.hex.int**.

so.write.real32

```
PROC so.write.real32 (CHAN OF SP fs, ts,
                      VAL REAL32 r,
                      VAL INT Ip, Dp)
```

Writes the value **r** (of type **REAL32**) to the screen as decimal ASCII digits. The number is padded out with leading spaces and an optional sign bit to the number of digits specified by **m** before and **n** after the decimal point. The total width of the number is **m** + **n** + 2, except in the cases described under **REAL32TOSTRING** (see section 24.7).

so.write.real64

```
PROC so.write.real64 (CHAN OF SP fs, ts,
                      VAL REAL64 r,
                      VAL INT Ip, Dp)
```

As **so.write.real32** but for 64-bit real numbers. Allows 3 digits for the exponent.

24.4.8 File output

These routines write characters and strings to a specified stream, usually a file. The result returned can take the values **spr.ok** or **spr.operation.failed**, which signal success or failure of the operation respectively.

Procedure	Parameter Specifiers
so.fwrite.char	CHAN OF SP fs, ts, VAL INT32 streamid, VAL BYTE char, BYTE result
so.fwrite.nl	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.fwrite.string	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result
so.fwrite.string.nl	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result
so.fwrite.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, field, BYTE result
so.fwrite.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT field, BYTE result
so.fwrite.hex.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, width, BYTE result
so.fwrite.hex.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT width, BYTE result
so.fwrite.real32	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL32 r, VAL INT Ip, Dp, BYTE result
so.fwrite.real64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL64 r, VAL INT Ip, Dp, BYTE result

Procedure definitions

so.fwrite.char

```
PROC so.fwrite.char (CHAN OF SP fs, ts,
                     VAL INT32 streamid,
                     VAL BYTE char,
                     BYTE result)
```

Writes a single character to the specified stream.

so.fwrite.nl

```
PROC so.fwrite.nl (CHAN OF SP fs, ts,
                   VAL INT32 streamid,
                   BYTE result)
```

Writes a newline sequence to the specified stream.

so.fwrite.string

```
PROC so.fwrite.string (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL []BYTE string,
                       BYTE result)
```

Writes a string to the specified stream.

so.fwrite.string.nl

```
PROC so.fwrite.string.nl (CHAN OF SP fs, ts,
                          VAL INT32 streamid,
                          VAL []BYTE string,
                          BYTE result)
```

As **so.fwrite.string**, but appends a newline sequence to the end of the string.

so.fwrite.int

```
PROC so.fwrite.int (CHAN OF SP fs, ts,
                    VAL INT32 streamid,
                    VAL INT n, field,
                    BYTE result)
```

Writes the value **n** (of type **INT**) to the specified stream as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified **field** width. If the **field** width is too small for the number it is widened as necessary; a zero value for **field** will give minimum width.

so.fwrite.int64

```
PROC so.fwrite.int64 (CHAN OF SP fs, ts,
                      VAL INT32 streamid,
                      VAL INT64 n, VAL INT field,
                      BYTE result)
```

As **so.fwrite.int** but for 64-bit integers. The **field** parameter behaves as in **so.fwrite.int**.

so.fwrite.hex.int

```
PROC so.fwrite.hex.int (CHAN OF SP fs, ts,
                        VAL INT32 streamid,
                        VAL INT n, width,
                        BYTE result)
```

Writes the value **n** (of type **INT**) to the specified stream as hexadecimal ASCII digits preceded by the '#' character. The number of characters to be printed is specified by **width**. If **width** is larger than the size of the number then the number is padded with leading '0's or 'F's as appropriate. If **width** is smaller than the size of the number, then the number is truncated to **width** digits.

so.fwrite.hex.int64

```
PROC so.fwrite.hex.int64 (CHAN OF SP fs, ts,
                          VAL INT32 streamid,
                          VAL INT64 n,
                          VAL INT width,
                          BYTE result)
```

As **so.fwrite.hex.int** but for 64-bit integers. The **width** parameter behaves as in **so.fwrite.hex.int**.

so.fwrite.real32

```
PROC so.fwrite.real32 (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL REAL32 r,
                       VAL INT Ip, Dp,
                       BYTE result)
```

Writes the value **r** (of type **REAL32**) to the specified stream as decimal ASCII digits. The number is padded out with leading spaces and an optional sign bit to the number of digits specified by **m** before and **n** after the decimal point. The total width of the number is **m** + **n** + 2, except in the cases described under **REAL32TOSTRING** (see section 24.7).

so.fwrite.real64

```
PROC so.fwrite.real64 (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL REAL64 r,
                       VAL INT Ip, Dp,
                       BYTE result)
```

As **so.fwrite.real32** but for 64-bit real numbers. Allows 3 digits for the exponent.

24.4.9 Miscellaneous commands

The miscellaneous group includes procedures for:

- Time and date processing
- Buffering and multiplexing
- AF-to-SP protocol conversion.

Time processing

Procedure	Parameter Specifiers
so.time.to.date	VAL INT32 input.time, [so.date.len]INT date
so.date.to.ascii	VAL [so.date.len]INT date, VAL BOOL long.years, VAL BOOL days.first, [so.time.string.len]BYTE string
so.time.to.ascii	VAL INT32 time, VAL BOOL long.years, VAL BOOL days.first [so.time.string.len]BYTE string
so.today.date	CHAN OF SP fs, ts, [so.date.len]INT date
so.today.ascii	CHAN OF SP fs, ts, VAL BOOL long.years, VAL BOOL days.first, [so.time.string.len]BYTE string

so.time.to.date

```
PROC so.time.to.date (VAL INT32 input.time,
                      [so.date.len]INT date)
```

Converts time (as supplied by **so.time**) to six integers, stored in the **date** array. The elements of the array are as follows:

Element of array	Data
0	Seconds past the minute
1	Minutes past the hour
2	The hour (24 hour clock)
3	The day of the month
4	The month
5	The year

so.date.to.ascii

```
PROC so.date.to.ascii
                    (VAL [so.date.len]INT date,
                     VAL BOOL long.years,
                     VAL BOOL days.first,
                     [so.time.string.len]BYTE string)
```

Converts an array of six integers containing the date (as supplied by **so.time.to.date**) into an ASCII string of the form:

HH:MM:SS DD/MM/YYYY

If **long.years** is FALSE then year field is reduced to two characters, the last two characters being padded with spaces. If **days.first** is FALSE then the ordering of day and month is changed (to the U.S. standard).

so.time.to.ascii

```
PROC so.time.to.ascii
                    (VAL INT32 time,
                     VAL BOOL long.years,
                     VAL BOOL days.first
                     [so.time.string.len]BYTE string)
```

Converts time (as supplied by **so.time**) into an ASCII string, as described for **so.date.to.ascii**.

so.today.date

```
PROC so.today.date (CHAN OF SP fs, ts,
                          [so.date.len]INT date)
```

Gives today's date as six integers, stored in the **data** array. The format of the array is the same as for **so.time.to.date**. If the date is unavailable all elements in **date** are set to zero.

so.today.ascii

```
PROC so.today.ascii
                   (CHAN OF SP fs, ts,
                    VAL BOOL long.years, days.first,
                    [so.time.string.len]BYTE string)
```

Gives today's date as an ASCII string, in the same format as procedure **so.date.to.ascii**. If the date is unavailable **string** is filled with spaces.

Buffers and multiplexors

This group of procedures are designed to assist with buffering and multiplexing data exchange between the program and host.

Procedure	Parameter Specifiers
so.buffer	CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper
so.overlapped.buffer	CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper
so.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper
so.overlapped.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper, []INT queue

so.buffer

```
PROC so.buffer (CHAN OF SP fs, ts,
                            from.user, to.user,
                CHAN OF BOOL stopper)
```

This procedure buffers data between the user and the host. It can be used by processes on a network to pass data to the host across intervening processes. It is terminated by sending a **FALSE** value on the channel **stopper**.

so.overlapped.buffer

```
PROC so.overlapped.buffer (CHAN OF SP fs, ts,
                                        from.user,
                           CHAN OF SP to.user,
                           CHAN OF BOOL stopper)
```

Similar to **so.buffer**, but contains built-in knowledge of host file server commands and allows many host communications to occur simultaneously through a train of processes. This can improve efficiency if the communications pass through many processes before reaching the server. It is terminated by a **FALSE** value on the channel **stopper**.

so.multiplexor

```
PROC so.multiplexor (CHAN OF SP fs, ts,
                     []CHAN OF SP from.user,
                                  to.user,
                     CHAN OF BOOL stopper)
```

This procedure multiplexes any number of pairs of **SP** protocol channels onto a single pair of **SP** protocol channels, which may go to the file server or another **SP** protocol multiplexor (or buffer). It is terminated by sending a **FALSE** value on the channel **stopper**.

so.overlapped.multiplexor

```
PROC so.overlapped.multiplexor
                    (CHAN OF SP fs, ts,
                     []CHAN OF SP from.user,
                                  to.user,
                     CHAN OF BOOL stopper,
                     []INT queue)
```

Similar to **so.multiplexor**, but can pipeline server requests. The number of requests than can be pipelined is determined by the size of **queue**, which must provide one word for each request that can be pipelined. Pipelining improves efficiency if the server requests have to pass through many processes on the way to and from the server. It is terminated by a **FALSE** value on the channel **stopper**.

Protocol converter

Procedure	Parameter Specifiers
af.to.sp	CHAN OF SP fs, ts, CHAN OF ANY from.user, to.user, VAL BOOL pass.terminate

af.to.sp

```
PROC af.to.sp (CHAN OF SP fs, ts,
               CHAN OF ANY from.user, to.user,
               VAL BOOL pass.terminate)
```

Converts channels from the AFSERVER protocol to the **SP** protocol. **fs** and **ts** are the **SP** channels from and to the host file server, and **from.user** and **to.user** are the channels between the application program and the AFSERVER. The boolean **pass.terminate** controls the action of the AFSERVER terminate command; if set to **TRUE** it will terminate the server, if set to **FALSE** it will not.

24.5 Streamio library

The streamio library contains routines for reading and writing to files and to the terminal at a higher level of abstraction than the hostio library. A set of constants for the streamio library is provided in the file **streamio.inc**.

The streamio routines can be classified into three main groups:

- Stream processes
- Stream input procedures
- Stream output procedures.

Stream input and output procedures are used to input and output characters in keystream **KS** and screen stream **SS** protocols. **KS** and **SS** protocols must be converted to the server protocol before communicating with the host.

Stream processes convert streams from keyboard or screen protocol to the server protocol **SP** or to related data structures. They are used to transfer data from the stream input and output routines to the host. Stream processes can be run as parallel processes serving stream input and output routines called in sequential code.

The key stream and screen stream protocols are identical to those used in the IMS D700D Transputer Development System (TDS) and facilitate the porting of programs between the TDS and the toolset.

24.5.1 Naming conventions

Procedures always begin with a prefix derived from the first parameter. Stream processes, where the SP channel (listed first) is used in combination with either the **KS** or **SS** protocols, are prefixed with '**so.**'. Stream input routines, which use only the **KS** protocol are prefixed with '**ks.**', and stream output routines, which use only the **SS** protocol, are prefixed with '**ss.**'. The single **KS** to **SS** conversion routine, which uses both protocols, is prefixed with '**ks.**'.

24.5.2 Stream processes

Procedure	Parameter Specifiers
so.keystream.from.kbd	CHAN OF SP fs, ts, CHAN OF KS keys.out, CHAN OF BOOL stopper, VAL INT ticks.per.poll
so.keystream.from.file	CHAN OF SP fs, ts, CHAN OF KS keys.out, VAL []BYTE filename, BYTE result
so.keystream.from.stdin	CHAN OF SP fs, ts, CHAN OF KS keys.out, BYTE result
ks.keystream.sink	CHAN OF KS keys
ks.keystream.to.scrstream	CHAN OF KS keyboard, CHAN OF SS scrn
ss.scrstream.sink	CHAN OF SS scrn
so.scrstream.to.file	CHAN OF SP fs, ts, CHAN OF SS scrn, VAL []BYTE filename, BYTE result
so.scrstream.to.stdout	CHAN OF SP fs, ts, CHAN OF SS scrn, BYTE result
ss.scrstream.to.array	CHAN OF SS scrn, []BYTE buffer
ss.scrstream.from.array	CHAN OF SS scrn, VAL []BYTE buffer
ss.scrstream.copy	CHAN OF SS scrn, scrn.out
ss.scrstream.fan.out	CHAN OF SS scrn, CHAN OF SS screen.out1, screen.out2
so.scrstream.to.ANSI	CHAN OF SP fs, ts, CHAN OF SS scrn
so.scrstream.to.TVI920	CHAN OF SP fs, ts, CHAN OF SS scrn

Procedure definitions

so.keystream.from.kbd

```
PROC so.keystream.from.kbd
                          (CHAN OF SP fs, ts,
                           CHAN OF KS keys.out,
                           CHAN OF BOOL stopper,
                           VAL INT ticks.per.poll)
```

Reads characters from the keyboard and outputs them one at a time as integers on the channel **keys.out**. It is terminated by sending FALSE on the boolean channel **stopper**. The procedure polls the keyboard at an interval determined by the value of **ticks.per.poll**, in transputer clock cycles. **ticks.per.poll** must not be zero.

After FALSE is sent on the channel **stopper** the procedure sends the negative value **ft.terminated** on **keys.out** to mark the end of the file.

so.keystream.from.file

```
PROC so.keystream.from.file
                          (CHAN OF SP fs, ts,
                           CHAN OF KS keys.out,
                           VAL []BYTE filename,
                           BYTE result)
```

As **so.keystream.from.kbd**, but reads characters from the specified file. Terminates automatically when it has reached the end of the file and all the characters have been read from the **keys.out** channel. The negative value **ft.terminated** is sent on the channel **keys.out** to mark the end of the file. The result value returned will be one of those returned by **so.gets**.

so.keystream.from.stdin

```
PROC so.keystream.from.stdin
                          (CHAN OF SP fs, ts,
                           CHAN OF KS keys.out,
                           BYTE result)
```

As **so.keystream.from.kbd**, but reads from the standard input stream. The standard input stream is normally assigned to the keyboard, but can be redirected by the host operating system.

ks.keystream.sink

```
PROC ks.keystream.sink  (CHAN OF KS keys)
```

Reads integers until **ft.terminated** is received, then terminates.

ks.keystream.to.scrstream

```
PROC ks.keystream.to.scrstream (CHAN OF KS
                                    keyboard,
                                CHAN OF SS scrn)
```

Converts key stream protocol to screen stream protocol.

ss.scrstream.sink

```
PROC ss.scrstream.sink (CHAN OF SS scrn)
```

Reads screen stream protocol and ignores it except for the negative value **ft.terminated** which terminates the procedure.

so.scrstream.to.file

```
PROC so.scrstream.to.file (CHAN OF SP fs, ts,
                           CHAN OF SS scrn,
                           VAL []BYTE filename,
                           BYTE result)
```

Creates a new file with the specified name and writes the data sent on channel **scrn** to it. The **scrn** channel uses the screen stream protocol which is used by all the stream output library routines (and is the same as the inmos TDS screen stream protocol). It terminates on receipt of the stream terminator from **ss.write.endstream**, or on an error condition. The error code returned by **result** can be any result returned by **so.write**.

If used in conjunction with **so.scrstream.fan.out** it may be used to file a copy of everything sent to the screen.

so.scrstream.to.stdout

```
PROC so.scrstream.to.stdout (CHAN OF SP fs, ts,
                             CHAN OF SS scrn,
                             BYTE result)
```

Performs the same operation as **so.scrstream.to.file**, but writes to the standard output stream. The standard output stream goes to the

screen, but can be redirected to a file by the host operating system.

ss.scrstream.to.array

```
PROC ss.scrstream.to.array (CHAN OF SS scrn,
                            []BYTE buffer)
```

Buffers a screen stream whose total size does not exceed the capacity of **buffer**, for debugging purposes or subsequent onward transmission using **so.scrstream.from.array**. The procedure terminates on receipt of the stream terminator from **ss.write.endstream**.

ss.scrstream.from.array

```
PROC ss.scrstream.from.array (CHAN OF SS scrn,
                              VAL []BYTE buffer)
```

Regenerates a screen stream buffered in **buffer** by a previous call of **so.scrstream.to.array**. Terminates when all buffered data has been sent.

ss.scrstream.fan.out

```
PROC ss.scrstream.fan.out
                         (CHAN OF SS scrn,
                          CHAN OF SS screen.out1,
                                     screen.out2)
```

Sends copies of everything received on the input channel **scrn** to two output channels. The procedure terminates on receipt of the stream terminator from **ss.write.endstream** without passing on the end-stream.

ss.scrstream.copy

```
PROC ss.scrstream.copy (CHAN OF SS scrn,
                                   scrn.out)
```

Copies screen stream protocol input on **scrn** to **scrn.out**. Terminates on receipt of the endstream terminator, which is not passed on.

so.scrstream.to.ANSI

```
PROC so.scrstream.to.ANSI (CHAN OF SP fs, ts,
                           CHAN OF SS scrn)
```

Converts screen stream protocol into a stream of BYTEs according to the requirements of ANSI terminal screen protocol. Not all of the screen stream commands are supported, as some are not straightforward to implement. Refer to the source of the procedure to determine which commands are supported. The procedure terminates on receipt of the stream terminator from **ss.write.endstream**.

so.scrstream.to.TVI920

```
PROC so.scrstream.to.TVI920 (CHAN OF SP fs, ts,
                             CHAN OF SS scrn)
```

Converts screen stream protocol into a stream of BYTEs according to the requirements of TVI920 (and compatible) terminals. Not all of the screen stream commands are supported, as some are not straightforward to implement. Refer to the source of the procedure to determine which commands are supported. The procedure terminates on receipt of the stream terminator from **ss.write.endstream**.

24.5.3 Stream input

These routines read characters and strings from the input stream, in **KS** protocol.

Procedure	Parameter Specifiers
ks.read.char	CHAN OF KS source, INT char
ks.read.line	CHAN OF KS source, INT len, []BYTE line, INT char
ks.read.int	CHAN OF KS source, INT number, char
ks.read.int64	CHAN OF KS source, INT64 number, INT char
ks.read.real32	CHAN OF KS source, REAL32 number, INT char
ks.read.real64	CHAN OF KS source, REAL64 number, INT char

Procedure definitions

ks.read.char

```
PROC ks.read.char (CHAN OF KS source, INT char)
```

Returns ASCII value of next **char** from source, (if input is from a file end of line is signified by the value **INT'*c'**).

ks.read.line

```
PROC ks.read.line (CHAN OF KS source, INT len,
                   []BYTE line, INT char)
```

Reads text into the array **line** up to and including **'*c'**, or up to and excluding any error code. Any **'*n'** encountered is thrown away. **len** is the length of the line. If there is an error its code is returned as **char**, otherwise the value of **char** will be **INT '*c'**. If the array is filled before a **'*c'** is encountered all further characters are ignored.

ks.read.int

```
PROC ks.read.int (CHAN OF KS source,
                  INT number, char)
```

Skips input up to a digit, **#**, **+** or **-**, then reads a sequence of digits to the first non-digit, returned as **char**, and converts the digits to an integer in **number**. **char** must be initialised to the first character of the number. If the first significant character is a **'#'** then a hexadecimal number is input, thereby allowing the user the option of which number base to use.

ks.read.int64

```
PROC ks.read.int64 (CHAN OF KS source,
                    INT64 number, INT char)
```

As **ks.read.int**, but for 64-bit integers.

ks.read.real32

```
PROC ks.read.real32 (CHAN OF KS source,
                     REAL32 number, INT char)
```

Skips input up to a digit, **+** or **-**, then reads a sequence of digits with optional decimal point and exponent) up to the first invalid character, returned as **char**. Converts the digits to a floating point value in **number**.

`ks.read.real64`

```
PROC ks.read.real64 (CHAN OF KS source,
                     REAL64 number, INT char)
```

As `ks.read.real32`, but for 64-bit real numbers.

24.5.4 Stream output

These routines write text, numbers and screen control codes to an output stream in SS protocol.

Procedure	Parameter Specifiers
`ss.write.char`	`CHAN OF SS scrn,` `VAL BYTE char`
`ss.write.nl`	`CHAN OF SS scrn`
`ss.write.string`	`CHAN OF SS scrn,` `VAL[]BYTE str`
`ss.write.endstream`	`CHAN OF SS scrn`
`ss.write.text.line`	`CHAN OF SS scrn,` `VAL []BYTE str`
`ss.write.int`	`CHAN OF SS scrn,` `VAL INT number, field`

`ss.write.int64`	`CHAN OF SS scrn,`
	`VAL INT64 number,`
	`VAL INT field`

`ss.write.hex.int`	`CHAN OF SS scrn,` `VAL INT number, field`

`ss.write.hex.int64`	`CHAN OF SS scrn,`
	`VAL INT64 number,`
	`VAL INT field`

Procedure	Parameter Specifiers
ss.write.real32	CHAN OF SS scrn, VAL REAL32 number, VAL INT Ip, Dp
ss.write.real64	CHAN OF SS scrn, VAL REAL64 number, VAL INT Ip, Dp
ss.goto.xy	CHAN OF SS scrn, VAL INT x, y
ss.clear.eol	CHAN OF SS scrn
ss.clear.eos	CHAN OF SS scrn
ss.beep	CHAN OF SS scrn
ss.up	CHAN OF SS scrn
ss.down	CHAN OF SS scrn
ss.left	CHAN OF SS scrn
ss.right	CHAN OF SS scrn
ss.insert.char	CHAN OF SS scrn, VAL BYTE ch
ss.delete.chr	CHAN OF SS scrn
ss.delete.chl	CHAN OF SS scrn
ss.ins.line	CHAN OF SS scrn
ss.del.line	CHAN OF SS scrn

Procedure definitions

`ss.write.char`

```
PROC ss.write.char (CHAN OF SS scrn,
                    VAL BYTE char)
```

Sends the ASCII value `char` down `scrn`, in `scrstream` protocol, to the current position in the output line.

`ss.write.nl`

```
PROC ss.write.nl (CHAN OF SS scrn)
```

Sends `"*c*n"` to the `scrn`.

ss.write.string

```
PROC ss.write.string (CHAN OF SS scrn,
                      VAL[]BYTE str)
```

Sends all characters in **str** to **scrn**.

ss.write.endstream

```
PROC ss.write.endstream (CHAN OF SS scrn)
```

Sends a special stream terminator value to **scrn**. A call of this is needed if **scrn** is a file interface, or other interface procedure without an explicit stopping channel, but not if it is a real screen channel.

ss.write.text.line

```
PROC ss.write.text.line (CHAN OF SS scrn,
                         VAL []BYTE str)
```

Sends a line of characters from **str**, optionally terminated by a **'*c'**, to **scrn**, followed by a line feed. This procedure should be used for text which is organised into complete lines.

ss.write.int

```
PROC ss.write.int (CHAN OF SS scrn,
                   VAL INT number, field)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified **field** width if necessary. If the number cannot be represented in **field** characters it is widened as necessary, a zero value for **field** will give minimum width. The converted number is sent to **scrn**.

ss.write.int64

```
PROC ss.write.int64 (CHAN OF SS scrn,
                     VAL INT64 number,
                     VAL INT field)
```

As **ss.write.int** but for 64-bit integers.

ss.write.hex.int

```
PROC ss.write.hex.int (CHAN OF SS scrn,
                       VAL INT number, field)
```

Converts **number** into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by **#**. The total number of characters sent is always **field + 1**, padding out with **0** or **F** on the left if necessary. The number is truncated at the left if the field is too narrow, thereby allowing the less significant part of any number to be printed. The converted number is sent to **scrn**.

ss.write.hex.int64

```
PROC ss.write.hex.int64 (CHAN OF SS scrn,
                         VAL INT64 number,
                         VAL INT field)
```

As **ss.write.hex.int** but for 64-bit integer values.

ss.write.real32

```
PROC ss.write.real32 (CHAN OF SS scrn,
                      VAL REAL32 number,
                      VAL INT Ip, Dp)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified number of digits **Ip** before and **Dp** after the decimal point. The converted number is sent to **scrn**.

The total width will be **Ip + Dp + 2** except in the cases described under **REAL32TOSTRING** (see section 24.7).

ss.write.real64

```
PROC ss.write.real64 (CHAN OF SS scrn,
                      VAL REAL64 number,
                      VAL INT Ip, Dp)
```

As for **ss.write.real32** but for 64-bit real values. Allows 3 digits for the exponent.

ss.goto.xy

```
PROC ss.goto.xy (CHAN OF SS scrn, VAL INT x, y)
```

Sends the cursor to screen position (x,y). The origin (0,0) is at the top left corner of the screen.

ss.clear.eol

```
PROC ss.clear.eol (CHAN OF SS scrn)
```

Clears from the cursor position to the end of the current screen line.

ss.clear.eos

```
PROC ss.clear.eos (CHAN OF SS scrn)
```

Clears from the cursor position to the end of the current line and all lines below.

ss.beep

```
PROC ss.beep (CHAN OF SS scrn)
```

Sends a bell code to the terminal.

ss.up

```
PROC ss.up (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one line up the screen.

ss.down

```
PROC ss.down (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one line down the screen.

ss.left

```
PROC ss.left (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one place left.

ss.right

```
PROC ss.right (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one place right.

ss.insert.char

```
PROC ss.insert.char (CHAN OF SS scrn,
                     VAL BYTE ch)
```

Sends a command to the terminal to move the character at the cursor and all those to the right of it one place to the right and inserts **char** at the cursor. The cursor moves one place right.

ss.delete.chr

```
PROC ss.delete.chr (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the character at the cursor and move the rest of the line one place to the left. The cursor does not move.

ss.delete.chl

```
PROC ss.delete.chl (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the character to the left of the cursor and move the rest of the line one place to the left. The cursor also moves one place left.

ss.ins.line

```
PROC ss.ins.line (CHAN OF SS scrn)
```

Sends a command to the terminal to move all lines below the current line down one line on the screen, losing the bottom line. The current line becomes blank.

ss.del.line

```
PROC ss.del.line (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the current line and move all lines below it up one line. The bottom line becomes blank.

24.6 String handling library

Library: `string.lib`

This library contains functions and procedures for handling strings and scanning lines of text. They assist with the manipulation of character strings such as names, commands, and keyboard responses.

The library provides routines for:

- Identifying characters
- Comparing strings
- Searching strings
- Editing strings
- Scanning lines of text

Result	Function	Parameter Specifiers
`BOOL`	`is.in.range`	`VAL BYTE char, bottom, top`
`BOOL`	`is.upper`	`VAL BYTE char`
`BOOL`	`is.lower`	`VAL BYTE char`
`BOOL`	`is.digit`	`VAL BYTE char`
`BOOL`	`is.hex.digit`	`VAL BYTE char`
`BOOL`	`is.id.char`	`VAL BYTE char`
`INT`	`compare.strings`	`VAL []BYTE str1, str2`
`BOOL`	`eqstr`	`VAL []BYTE s1,s2`
`INT`	`string.pos`	`VAL []BYTE search, str`
`INT`	`char.pos`	`VAL BYTE search,` `VAL []BYTE str`
`INT, BYTE`	`search.match`	`VAL []BYTE possibles, str`
`INT, BYTE`	`search.no.match`	`VAL []BYTE possibles, str`

Procedure	Parameter Specifiers
str.shift	[]BYTE str, VAL INT start, len, shift, BOOL not.done
delete.string	INT len, []BYTE str, VAL INT start, size, BOOL not.done
insert.string	VAL []BYTE new.str, INT len, []BYTE str, VAL INT start, BOOL not.done
to.upper.case	[]BYTE str
to.lower.case	[]BYTE str
append.char	INT len, []BYTE str, VAL BYTE char
append.text	INT len, []BYTE str, VAL []BYTE text
append.int	INT len, []BYTE str, VAL INT number, field
append.int64	INT len, []BYTE str, VAL INT64 number, VAL INT field
append.hex.int	INT len, []BYTE str, VAL INT number, field
append.hex.int64	INT len, []BYTE str, VAL INT64 number, VAL INT width
append.real32	INT len, []BYTE str, VAL REAL32 number, VAL INT Ip, Dp
append.real64	INT len, []BYTE str, VAL REAL64 number, VAL INT Ip, Dp

Procedure	Parameter Specifiers
next.word.from.line	VAL []BYTE line, INT ptr, INT len, []BYTE word, BOOL ok
next.int.from.line	VAL []BYTE line, INT ptr, number, BOOL ok

24.6.1 Character identification

is.in.range

```
BOOL FUNCTION is.in.range (VAL BYTE char, bottom,
                           top)
```

Returns **TRUE** if the value of **char** is in the range defined by **bottom** and **top** inclusive.

is.upper

```
BOOL FUNCTION is.upper (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII upper case letter.

is.lower

```
BOOL FUNCTION is.lower (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII lower case letter.

is.digit

```
BOOL FUNCTION is.digit (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII decimal digit.

is.hex.digit

```
BOOL FUNCTION is.hex.digit (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII hexadecimal digit. Upper or lower case letters A–F are allowed.

is.id.char

```
BOOL FUNCTION is.id.char (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII character which can be part of an occam name.

24.6.2 String comparison

These two procedures allow strings to be compared for order or for equality.

compare.strings

```
INT FUNCTION compare.strings (VAL []BYTE str1,
                              str2)
```

This general purpose ordering function compares two strings according to the lexicographic ordering standard. (Lexicographic ordering is the ordering used in dictionaries etc., using the ASCII values of the bytes). It returns one of the 5 results 0, 1, −1, 2, −2 as follows.

0 The strings are exactly the same in length and content.

1 **str2** is a leading sub-string of **str1**

−1 **str1** is a leading sub-string of **str2**

2 **str1** is lexicographically later than **str2**

−2 **str2** is lexicographically later than **str1**

So if **s** is 'abcd':

```
compare.strings ("abc", [s FROM 0 FOR 3]) = 0
compare.strings ("abc", [s FROM 0 FOR 2]) = 1
compare.strings ("abc", s)                = -1
compare.strings ("bc", s)                 = 2
compare.strings ("a4", s)                 = -2
```

eqstr

```
BOOL FUNCTION eqstr (VAL []BYTE s1,s2)
```

This is an optimised test for string equality. It returns **TRUE** if the two strings are the same size and have the same contents.

24.6.3 String searching

These procedures allow a string to be searched for a match with a single byte or a string of bytes, for a byte which is one of a set of possible bytes, or for a byte which is *NOT* one of a set of bytes. Searches insensitive to alphabetic case should use **to.upper.case** or **to.lower.case** on both operands before using these procedures.

string.pos

```
INT FUNCTION string.pos (VAL []BYTE search, str)
```

Returns the position in **str** of the first occurrence of a sub-string which exactly matches **search**. Returns −1 if there is no such match.

char.pos

```
INT FUNCTION char.pos (VAL BYTE search,
                       VAL []BYTE str)
```

Returns the position in **str** of the first occurrence of the byte **search**. Returns −1 if there is no such byte.

search.match

```
INT, BYTE FUNCTION search.match
                   (VAL []BYTE possibles, str)
```

Searches **str** for any one of the bytes in the array **possibles**. If one is found its index and identity are returned as results. If none is found then −1, 255(BYTE) are returned.

search.no.match

```
INT, BYTE FUNCTION search.no.match
                   (VAL []BYTE possibles, str)
```

Searches **str** for a byte which does not match any one of the bytes in the array **possibles**. If one is found its index and identity are returned as results. If none is found then −1, 255(BYTE) are returned.

24.6.4 String editing

These procedures allow strings to be edited. The string to be edited is stored in an array which may contain unused space. The editing operations supported are: deletion of a number of characters and the closing of the gap created;

insertion of a new string starting at any position within a string, which creates a gap of the necessary size.

These two operations are supported by a lower level procedure for shifting a consecutive substring left or right within the array. The lower level procedure does exhaustive tests against overflow.

str.shift

```
PROC str.shift ([]BYTE str, VAL INT start,
                len, shift, BOOL not.done)
```

Take a substring **[str FROM start FOR len]**, and copies it to a position **shift** places to the right. Any implied actions involving bytes outside the string are not performed and cause the error flag **not.done** to be set **TRUE**. Negative values of **shift** cause leftward moves.

delete.string

```
PROC delete.string (INT len, []BYTE str,
                    VAL INT start, size,
                    BOOL not.done)
```

Deletes **size** bytes from the string **str** starting at **str[start]**. There are initially **len** significant characters in **str** and it is decremented appropriately. If **start** is outside the string, or **size** is negative or greater than the string length, then no action occurs and **not.done** is set **TRUE**.

insert.string

```
PROC insert.string (VAL []BYTE new.str, INT len,
                    []BYTE str, VAL INT start,
                    BOOL not.done)
```

Creates a gap in **str** before **str[start]** and copies the string **new.str** into it. There are initially **len** significant characters in **str** and **len** is incremented by the length of **new.str** inserted. Any overflow of the declared SIZE of **str** results in truncation at the right and setting **not.done** to **TRUE**. This procedure may be used for simple concatenation on the right by setting **start = len** or on the left by setting **start = 0**. This method of concatenation differs from that using the **append.** procedures in that it can never cause the program to stop.

to.upper.case

```
PROC to.upper.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to upper case.

to.lower.case

```
PROC to.lower.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to lower case.

append.char

```
PROC append.char (INT len, []BYTE str,
                  VAL BYTE char)
```

Writes a byte **char** into the array **str** at **str[len]**. **len** is incremented by 1. Behaves like **STOP** if the array overflows.

append.text

```
PROC append.text (INT len, []BYTE str,
                  VAL []BYTE text)
```

Writes a string **text** into the array **str**, starting at **str[len]** and computing a new value for **len**. Behaves like **STOP** if the array overflows.

append.int

```
PROC append.int (INT len, []BYTE str,
                 VAL INT number, field)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified **field** width if necessary. If the number cannot be represented in **field** characters it is widened as necessary. A zero value for **field** will give minimum width. The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows.

append.int64

```
PROC append.int64 (INT len, []BYTE str,
                   VAL INT64 number,
                   VAL INT field)
```

As **append.int** but for 64-bit integers.

append.hex.int

```
PROC append.hex.int (INT len, []BYTE str,
                     VAL INT number, width)
```

Converts **number** into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by **#**. The total number of characters sent is always **width+1**, padding out with **0** or **F** on the left if necessary. The number is truncated at the left if the field is too narrow, thereby allowing the less significant part of any number to be printed. The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows.

append.hex.int64

```
PROC append.hex.int64 (INT len, []BYTE str,
                       VAL INT64 number,
                       VAL INT width)
```

As **append.hex.int** but for 64-bit integers.

append.real32

```
PROC append.real32 (INT len, []BYTE str,
                    VAL REAL32 number,
                    VAL INT Ip, Dp)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified number of digits **Ip** before and **Dp** after the decimal point. The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows.

The total width will be **Ip + Dp + 2** except in the cases described under **REAL32TOSTRING** (see section 24.7).

append.real64

```
PROC append.real64 (INT len, []BYTE str,
                    VAL REAL64 number,
                    VAL INT Ip, Dp)
```

As **append.real32**, but for 64-bit real values. Allows 3 digits for the exponent.

24.6.5 Line parsing

These two procedures read a line serially, returning the next word and next integer respectively.

next.word.from.line

```
PROC next.word.from.line (VAL []BYTE line,
                          INT ptr, INT len,
                          []BYTE word,
                          BOOL ok)
```

Skips leading spaces and tabs and reads the next word from the string **line**. The value of **ptr** is the starting point of the search for the word.

If the first non-space and non-tab character found is not a printable ASCII character, or if the end of the string is reached, the boolean **ok** is set to FALSE. If the word is followed by a space, tab, or the end of the string, **ok** is also set to TRUE, otherwise it is set to FALSE.

The pointer **ptr** is updated to point either to the position beyond the last character read, or to the end of the string, whatever value of **ok** is set.

next.int.from.line

```
PROC next.int.from.line (VAL []BYTE line,
                         INT ptr, number,
                         BOOL ok)
```

Skips leading spaces and tabs and reads the next integer from the string **line**. The value of **ptr** is the starting point of the search.

If the first non-space and non-tab character found is not a digit or sign (+ or -), or if the end of the string is reached, the boolean **ok** is set to FALSE. If the integer is followed by a space, tab, or the end of the string, **ok** is also set to TRUE, otherwise it is set to FALSE.

The pointer **ptr** is updated to point either to the position beyond the last character read, or to the end of the string, whatever value of **ok** is set.

24.7 Type conversion library

Library: `convert.lib`

This library contains procedures for converting numeric variables to strings and vice versa.

String to numeric conversions return two results, the converted value and a boolean error indication. Numeric to string conversions return the converted string and an integer which represents the number of significant characters written into the string.

Procedure	Parameter Specifiers
INTTOSTRING	INT len, []BYTE string, VAL INT n
INT16TOSTRING	INT len, []BYTE string, VAL INT16 n
INT32TOSTRING	INT len, []BYTE string, VAL INT32 n
INT64TOSTRING	INT len, []BYTE string, VAL INT64 n
HEXTOSTRING	INT len, []BYTE string, VAL INT n
HEX16TOSTRING	INT len, []BYTE string, VAL INT16 n
HEX32TOSTRING	INT len, []BYTE string, VAL INT32 n
HEX64TOSTRING	INT len, []BYTE string, VAL INT64 n
REAL32TOSTRING	INT len, []BYTE string, VAL REAL32 X, VAL INT Ip, Dp
REAL64TOSTRING	INT len, []BYTE string, VAL REAL64 X, VAL INT Ip, Dp
BOOLTOSTRING	INT len, []BYTE string, VAL BOOL b
STRINGTOINT	BOOL Error, INT n, VAL []BYTE string
STRINGTOINT16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOINT32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOINT64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOHEX	BOOL Error, INT n, VAL []BYTE string
STRINGTOHEX16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOHEX32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOHEX64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOREAL32	BOOL Error, REAL32 X, VAL []BYTE string
STRINGTOREAL64	BOOL Error, REAL64 X, VAL []BYTE string
STRINGTOBOOL	BOOL Error, b, VAL []BYTE string

24.7.1 Procedure definitions

INTTOSTRING

```
PROC INTTOSTRING (INT len, []BYTE string,
                  VAL INT n)
```

Converts an integer value to a string.

INT16TOSTRING

```
PROC INT16TOSTRING (INT len, []BYTE string,
                    VAL INT16 n)
```

As **INTTOSTRING** but for 16-bit integers.

INT32TOSTRING

```
PROC INT32TOSTRING (INT len, []BYTE string,
                    VAL INT32 n)
```

As **INTTOSTRING** but for 32-bit integers.

INT64TOSTRING

```
PROC INT64TOSTRING (INT len, []BYTE string,
                    VAL INT64 n)
```

As **INTTOSTRING** but for 64-bit integers.

HEXTOSTRING

```
PROC HEXTOSTRING (INT len, []BYTE string,
                  VAL INT n)
```

Converts a hexadecimal integer value to a string.

HEX16TOSTRING

```
PROC HEX16TOSTRING (INT len, []BYTE string,
                    VAL INT16 n)
```

As **HEXTOSTRING** but for 16-bit integers.

HEX32TOSTRING

```
PROC HEX32TOSTRING (INT len, []BYTE string,
                    VAL INT32 n)
```

As HEXTOSTRING but for 32-bit integers.

HEX64TOSTRING

```
PROC HEX64TOSTRING (INT len, []BYTE string,
                    VAL INT64 n)
```

As HEXTOSTRING but for 64-bit integers.

REAL32TOSTRING

```
PROC REAL32TOSTRING (INT len, []BYTE string,
                     VAL REAL32 X,
                     VAL INT Ip, Dp)
```

Converts a 32-bit real number to a string. The total width will be `Ip + Dp + 2` except in the following cases:

If the value will not fit, an exponential form is used.

If `Ip` is zero, an exponential form with `Dp` significant digits is used, giving a field width of `Dp + 6`.

If `Ip` and `Dp` are zero, a minimum field width free format is used.

Numbers which correspond to the IEEE standard concepts of 'Infinity' and 'NotaNumber' produce the texts `Inf` and `NaN`, respectively.

In exponential forms a number in the range [1.0, 10.0] is followed by `E`, a + or − sign, and a 2 digit decimal exponent.

REAL64TOSTRING

```
PROC REAL64TOSTRING (INT len, []BYTE string,
                     VAL REAL64 X,
                     VAL INT Ip, Dp)
```

As REAL32TOSTRING but for 64-bit numbers.

BOOLTOSTRING

```
PROC BOOLTOSTRING (INT len, []BYTE string,
                   VAL BOOL b)
```

Converts a boolean value to a string.

STRINGTOINT

```
PROC STRINGTOINT (BOOL Error, INT n,
                  VAL []BYTE string)
```

Converts a string to a decimal integer.

STRINGTOINT16

```
PROC STRINGTOINT16 (BOOL Error, INT16 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 16-bit integer.

STRINGTOINT32

```
PROC STRINGTOINT32 (BOOL Error, INT32 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 32-bit integer.

STRINGTOINT64

```
PROC STRINGTOINT64 (BOOL Error, INT64 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 64-bit integer.

STRINGTOHEX

```
PROC STRINGTOHEX (BOOL Error, INT n,
                  VAL []BYTE string)
```

Converts a string to a hexadecimal integer.

STRINGTOHEX16

```
PROC STRINGTOHEX16 (BOOL Error, INT16 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 16-bit integer.

STRINGTOHEX32

```
PROC STRINGTOHEX32 (BOOL Error, INT32 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 32-bit integer.

STRINGTOHEX64

```
PROC STRINGTOHEX64 (BOOL Error, INT64 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 64-bit integer.

STRINGTOREAL32

```
PROC STRINGTOREAL32 (BOOL Error, REAL32 X,
                     VAL []BYTE string)
```

Converts a string to a 32-bit real number.

STRINGTOREAL64

```
PROC STRINGTOREAL64 (BOOL Error, REAL64 X,
                     VAL []BYTE string)
```

As **STRINGTOREAL32** but converts to a 64-bit number.

STRINGTOBOOL

```
PROC STRINGTOBOOL (BOOL Error, b,
                   VAL []BYTE string)
```

Converts a string to a boolean value.

24.8 Block CRC library

Library: `crc.lib`

The block CRC library provides two functions for generating CRC codes from byte strings. The application that uses the function must also provide the old CRC code.

For further infomation about CRC functions see '*INMOS Technical note 26: Notes on graphics support and performance improvements on the IMS T800*'.

Result	Function	Parameter Specifiers

`INT`	`CRCFROMMSB`	`VAL []BYTE InputString,`
		`VAL INT PolynomialGenerator,`
		`INT OldCRC`

`INT`	`CRCFROMLSB`	`VAL []BYTE InputString`
		`VAL INT PolynomialGenerator,`
		`INT OldCRC`

24.8.1 Function definitions

`CRCFROMMSB`

```
INT FUNCTION CRCFROMMSB (VAL []BYTE InputString,
                         VAL INT PolynomialGenerator,
                         INT OldCRC)
```

The string of bytes is polynomially divided by the generator, starting at the most significant bit of the most significant byte.

`CRCFROMLSB`

```
INT FUNCTION CRCFROMLSB (VAL []BYTE InputString,
                         VAL INT PolynomialGenerator,
                         INT OldCRC)
```

The string of bytes is polynomially divided by the generator, starting at the least significant bit of the least significant byte.

24.9 Extraordinary link handling library

Library: xlink.lib

The extraordinary link handling library contains routines for handling communication failures errors on a link. Four procedures are provided to allow failures on input and output channels to be handled by timeout or by signalling the failure on another channel. A fifth procedure allows the channel to be reset.

Procedure	Parameter Specifiers
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
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
Reinitialise	CHAN OF ANY c

24.9.1 Procedure definitions

The four 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 for the message allows an object of any type to be passed along the channel providing it is retyped first.

InputOrFail.t

```
PROC InputOrFail.t (CHAN OF ANY c, []BYTE mess,
                    TIMER TIME,
                    VAL INT t, BOOL aborted)
```

This procedure is used for communication where failure is detected by a timeout. It takes a timer parameter **TIME**, and an absolute time **t**. The procedure treats the communication as having failed when the time as measured by the timer **TIME** is **AFTER** the specified time **t**.

OutputOrFail.t

```
PROC OutputOrFail.t (CHAN OF ANY c,
                     VAL []BYTE mess,
                     TIMER TIME,
                     VAL INT t, BOOL aborted)
```

This procedure is used for communication where failure is detected by a timeout. It takes a timer parameter **TIME**, and an absolute time **t**. The procedure treats the communication as having failed when the time as measured by the timer **TIME** is **AFTER** the specified time **t**.

InputOrFail.c

```
PROC InputOrFail.c (CHAN OF ANY c, []BYTE mess,
                    CHAN OF INT kill,
                    BOOL aborted)
```

This procedure provides communication where failure cannot be detected by a simple timeout. In this case failure must be signalled to the inputting procedure via a message on the channel **kill**. The kill message is of type **INT** and can be any value.

OutputOrFail.c

```
PROC OutputOrFail.c (CHAN OF ANY c,
                     VAL []BYTE mess,
                     CHAN OF INT kill,
                     BOOL aborted)
```

This procedure provides 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** and can be any value.

Reinitialise

```
PROC Reinitialise (CHAN OF ANY c)
```

This procedure may be used to reinitialise the link channel **c** after it is known that all activity on the link has ceased.

Reinitialise 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.

24.10 Process library

Library: process.lib

The process library provides two procedures. The first supports processes running on boot from ROM transputer boards, and the second is a timer process for analysing deadlocks. The process library is supplied in object and source forms.

Procedure	Parameter Specifiers
ss.B00x.stream.driver	CHAN OF SS from.user.screen, CHAN OF KS to.user.kbd, VAL INT board.type, port, baud.rate, screen.type
debug.timer	CHAN OF INT stop

24.10.1 Procedure definitions

ss.B00x.stream.driver

```
PROC ss.B00x.stream.driver
           (CHAN OF SS from.user.screen,
            CHAN OF KS to.user.kbd,
            VAL INT board.type, port,
            baud.rate, screen.type)
```

This interface procedure may be run in parallel with any application on an IMS B001, IMS B002, or IMS B006 transputer evaluation board. It takes input in screen stream protocol on the channel **from.user.scrn** and sends it to an RS232 output, and sends the corresponding input in key stream protocol to the channel **to.user.kbd**.

board.type should be 1 for B001, 2 for B002, or 6 for B006. Any other value will reset the error flag. The uart port is defined by passing 0 (terminal) or 1 (host) in the parameter **port**.

If **baud.rate** is non-zero the UART is reset when the procedure starts executing. The value passed should be one of: 38400, 19200, 9600, 7200, 4800, 2400, 2000, 1800, 1200, 1050, 600, 300, 200, 150, 134, 110, 75 or 50 for B001 and B002 boards, and any of the same values except 38400 and 19200 for B006 boards. If **baud.rate** is zero the reset is assumed to have been already performed (e.g. by code in the ROM).

The procedure supports ANSI and TVI920 screen protocols. **screen.type** should be set to 0 (zero) for ANSI based terminals (for example the VT100), and 1 (one) for TVI920 based terminals.

debug.timer

```
PROC debug.timer (CHAN OF INT stop)
```

A timer process for use when analysing deadlocks in occam programs. Section 7.4 gives an example of how to debug a deadlocked program using similar code.

The parameters are as follows:

stop A single **INT** value on this channel terminates the timer process.

Appendices

A Names defined by the software

All names which may appear in occam source text and which are defined either by the language, the compiler or the libraries are given below in alphabetical order.

Toolset constants are not included; for listings of the constants files see appendix C.

The names in this table are grouped into the following classes:

1 *Language keyword.* Keyword defined in the language reference manual.

2 *Compiler keyword.* Keyword defined by the current compiler implementation.

3 *Compiler predefine.* A procedure or function which is predeclared by the compiler and implemented by in line code.

4 *Compiler library.* A library procedure or function which is used by compiler generated code. On some processors these are implemented by a function in a library with the name indicated, on others they are implemented as in line code (a predefine). Where a compiler library routine is implemented as a predefine for some transputers the word *predefine* appears in the 'Notes' column.

5 *System library.* Library routines for special transputer system operations. Consists of the libraries **crc.lib** and **xlink.lib**.

6 *Maths library.* A function in the elementary function libraries. The library name depends on version required (single or double length).

7 *Maths utility library.* Supporting function for maths library.

8 *Io library.* A procedure or function in the input/output and supporting libraries (**hostio.lib**, **streamio.lib**, **string.lib**, and **convert.lib**). The library name which must be used to access it is also shown.

9 *Compiler input directive.* A word in occam source code recognised by the compilation system for special action at compile time. Always preceded in occam source by the character '#'.

Any name which is not a language keyword may be redeclared as an identifier in

an occam program. However, redefining a name of a compiler library procedure or function can have unexpected consequences and it is strongly recommended that all the names in these tables are reserved for the uses specified.

Name	Class	Library	Notes
ABS	compiler library		predefine
ACOS	maths library	snglmath	also tbmaths
af.to.sp	io library	hostio	
AFTER	language keyword		
ALOG	maths library	snglmath	also tbmaths
ALOG10	maths library	snglmath	also tbmaths
ALT	language keyword		
AND	language keyword		
ANY	language keyword		
append.char	io library	string	
append.hex.int64	io library	string	
append.hex.int	io library	string	
append.int64	io library	string	
append.int	io library	string	
append.real32	io library	string	
append.real64	io library	string	
append.text	io library	string	
ARGUMENT.REDUCE	compiler library		
ASHIFTLEFT	compiler predefine		
ASHIFTRIGHT	compiler predefine		
ASIN	maths library	snglmath	also tbmaths
AT	language keyword		
ATAN	maths library	snglmath	also tbmaths
ATAN2	maths library	snglmath	also tbmaths
BITAND	language keyword		
BITCOUNT	compiler library		predefine
BITNOT	language keyword		
BITOR	language keyword		
BITREVNBITS	compiler library		predefine
BITREVWORD	compiler library		predefine
BOOL	language keyword		
BOOLTOSTRING	io library	convert	
BYTE	language keyword		
CASE	language keyword		
CAUSEERROR	compiler predefine		
CHAN	language keyword		
char.pos	io library	string	

Name	Class	Library	Notes
CLIP2D	compiler library		predefine
COMMENT	compiler directive		
compare.strings	io library	string	
COPYSIGN	compiler library		
COS	maths library	snglmath	also tbmaths
COSH	maths library	snglmath	also tbmaths
CRCBYTE	compiler library		predefine
CRCFROMLSB	system library	crc	
CRCFROMMSB	system library	crc	
CRCWORD	compiler library		predefine
DABS	compiler library		predefine
DACOS	maths library	dblmath	also tbmaths
DALOG	maths library	dblmath	also tbmaths
DALOG10	maths library	dblmath	also tbmaths
DARGUMENT.REDUCE	compiler library		
DASIN	maths library	dblmath	also tbmaths
DATAN	maths library	dblmath	also tbmaths
DATAN2	maths library	dblmath	also tbmaths
DCOPYSIGN	compiler library		
DCOS	maths library	dblmath	also tbmaths
DCOSH	maths library	dblmath	also tbmaths
DDIVBY2	compiler library		predefine
delete.string	io library	string	
DEXP	maths library	dblmath	also tbmaths
DFLOATING.UNPACK	compiler library		
DFPINT	compiler library		predefine
DIEEECOMPARE	compiler library		
DISNAN	compiler library		predefine
DIVBY2	compiler library		predefine
DLOGB	compiler library		
DMINUSX	compiler library		predefine
DMULBY2	compiler library		predefine
DNEXTAFTER	compiler library		
DNOTFINITE	compiler library		predefine
DORDERED	compiler library		predefine
DPOWER	maths library	dblmath	also tbmaths
DRAN	maths library	dblmath	also tbmaths
DRAW2D	compiler library		predefine
DSCALEB	compiler library		
DSIN	maths library	dblmath	also tbmaths

Name	Class	Library	Notes
DSINH	maths library	dblmath	also tbmaths
DSQRT	compiler library		predefine
DTAN	maths library	dblmath	also tbmaths
DTANH	maths library	dblmath	also tbmaths
ELSE	language keyword		
eqstr	io library	string	
EXP	maths library	snglmath	also tbmaths
FALSE	language keyword		
FIX	maths utility library		
FIX64	maths utility library		
FLOATING.UNPACK	compiler library		
FOR	language keyword		
FPINT	compiler library		predefine
FracDiv	maths utility library		
FracDiv64	maths utility library		
FRACMUL	compiler predefine		T4,T8,T425, TA,TB,TC
FracMult64	maths utility library		
FROM	language keyword		
FUNCTION	language keyword		
GUY	compiler keyword		
HEX16TOSTRING	io library	convert	
HEX32TOSTRING	io library	convert	
HEX64TOSTRING	io library	convert	
HEXTOSTRING	io library	convert	
IEEE32OP	compiler library		
IEEE32REM	compiler library		not T8
IEEE64OP	compiler library		
IEEE64REM	compiler library		not T8
IEEECOMPARE	compiler library		
IF	language keyword		
IMPORT	compiler directive		
IN	language keyword		
IncExp	maths utility library		
IncExp64	maths utility library		
INCLUDE	compiler directive		
InputOrFail.c	system library	xlink	
InputOrFail.t	system library	xlink	
insert.string	io library	string	
INT	language keyword		

Name	Class	Library	Notes
INT16	language keyword		
INT16ADD	compiler library		not T2
INT16BITAND	compiler library		not T2
INT16BITNOT	compiler library		not T2
INT16BITOR	compiler library		not T2
INT16DIV	compiler library		not T2
INT16EQ	compiler library		not T2
INT16GT	compiler library		not T2
INT16LSHIFT	compiler library		not T2
INT16MINUS	compiler library		not T2
INT16MUL	compiler library		not T2
INT16PLUS	compiler library		not T2
INT16REM	compiler library		not T2
INT16RSHIFT	compiler library		not T2
INT16SUB	compiler library		not T2
INT16TIMES	compiler library		not T2
INT16TOINT32	compiler library		T2 only
INT16TOINT64	compiler library		T2 only
INT16TOREAL32	compiler library		T2 only
INT16TOREAL64	compiler library		T2 only
INT16TOSTRING	io library	convert	
INT16XOR	compiler library		not T2
INT32	language keyword		
INT32ADD	compiler library		T2 only
INT32BITAND	compiler library		T2 only
INT32BITNOT	compiler library		T2 only
INT32BITOR	compiler library		T2 only
INT32DIV	compiler library		T2 only
INT32DIVREM	compiler library		T2 only
INT32EQ	compiler library		T2 only
INT32GT	compiler library		T2 only
INT32LSHIFT	compiler library		T2 only
INT32MINUS	compiler library		T2 only
INT32MUL	compiler library		T2 only
INT32PLUS	compiler library		T2 only
INT32REM	compiler library		T2 only
INT32RSHIFT	compiler library		T2 only
INT32SUB	compiler library		T2 only
INT32TIMES	compiler library		T2 only
INT32TOINT16	compiler library		T2 only

Name	Class	Library	Notes
INT32TOINT64	compiler library		
INT32TOREAL32	compiler library		not T8
INT32TOREAL64	compiler library		not T8
INT32TOSTRING	io library	convert	
INT32XOR	compiler library		T2 only
INT64	language keyword		
INT64ADD	compiler library		
INT64BITAND	compiler library		
INT64BITNOT	compiler library		
INT64BITOR	compiler library		
INT64DIV	compiler library		
INT64DIVREM	compiler library		
INT64EQ	compiler library		
INT64GT	compiler library		
INT64LSHIFT	compiler library		
INT64MINUS	compiler library		
INT64MUL	compiler library		
INT64PLUS	compiler library		
INT64REM	compiler library		
INT64RSHIFT	compiler library		
INT64SUB	compiler library		
INT64TIMES	compiler library		
INT64TOINT16	compiler library		T2 only
INT64TOINT32	compiler library		
INT64TOREAL32	compiler library		not T8
INT64TOREAL64	compiler library		not T8
INT64TOSTRING	io library	convert	
INT64XOR	compiler library		
INTTOSTRING	io library	convert	
io.handler	io library	process	
IS	language keyword		
is.digit	io library	string	
is.hex.digit	io library	string	
is.id.char	io library	string	
is.in.range	io library	string	
is.lower	io library	string	
is.upper	io library	string	
ISNAN	compiler library		predefine
KERNEL.RUN	compiler predefine		

Name	Class	Library	Notes
ks.keystream.sink	io library	streamio	
ks.keystream.to.scrstream	io library	streamio	
ks.read.char	io library	streamio	
ks.read.hex.int	io library	streamio	
ks.read.hex.int64	io library	streamio	
ks.read.int	io library	streamio	
ks.read.int64	io library	streamio	
ks.read.line	io library	streamio	
ks.read.real32	io library	streamio	
ks.read.real64	io library	streamio	

Name	Class	Libr	Notes
LOAD.BYTE.VECTOR	compiler predefine		
LOAD.INPUT.CHANNEL	compiler predefine		
LOAD.INPUT.CHANNEL.VECTOR	compiler predefine		
LOAD.OUTPUT.CHANNEL	compiler predefine		
LOAD.OUTPUT.CHANNEL.VECTOR	compiler predefine		
LOGB	compiler library		
LONGADD	compiler predefine		
LONGDIFF	compiler predefine		
LONGDIV	compiler predefine		
LONGPROD	compiler predefine		
LONGSUB	compiler predefine		
LONGSUM	compiler predefine		
MINUS	language keyword		
MINUSX	compiler library		predefine
MOSTNEG	language keyword		
MOSTPOS	language keyword		
MOVE2D	compiler library		predefine
MULBY2	compiler library		predefine
next.int.from.line	io library	string	
next.word.from.line	io library	string	
NEXTAFTER	compiler library		
NORMALISE	compiler predefine		
NORMALISE64	maths utility library		
NOT	language keyword		
NOTFINITE	compiler library		predefine
OF	language keyword		
OPTION	compiler directive		

Name	Class	Library	Notes
OR	language keyword		
ORDERED	compiler library		predefine
OutputOrFail.c	system library	xlink	
OutputOrFail.t	system library	xlink	
PAR	language keyword		
PLACE	language keyword		
PLACED	language keyword		
PLUS	language keyword		
PORT	language keyword		
POWER	maths library	snglmath	also tbmaths
PRI	language keyword		
PROC	language keyword		
PROCESSOR	language keyword		
PROTOCOL	language keyword		
QRealIDiv	compiler library		T2 only
QRealIMul	compiler library		T2 only
QUADNORMALISE	compiler library		T2 only
QUADSHIFTLEFT	compiler library		T2 only
QUADSHIFTRIGHT	compiler library		T2 only
RAN	maths library	snglmath	also tbmaths
REAL32	language keyword		predefine
REAL32EQ	compiler library		predefine
REAL32EQERR	compiler library		predefine
REAL32GT	compiler library		predefine
REAL32GTERR	compiler library		predefine
REAL32OP	compiler library		
REAL32OPERR	compiler library		predefine
REAL32REM	compiler library		
REAL32REMERR	compiler library		predefine
REAL32TOINT16	compiler library		
REAL32TOINT32	compiler library		
REAL32TOINT64	compiler library		
REAL32TOREAL64	compiler library		
REAL32TOSTRING	io library	convert	
REAL64	language keyword		
REAL64EQ	compiler library		predefine
REAL64EQERR	compiler library		predefine
REAL64GT	compiler library		predefine
REAL64GTERR	compiler library		predefine
REAL64OP	compiler library		

Name	Class	Library	Notes
REAL64OPERR	compiler library		predefine
REAL64REM	compiler library		
REAL64REMERR	compiler library		predefine
REAL64TOINT16	compiler library		
REAL64TOINT32	compiler library		predefine
REAL64TOINT64	compiler library		predefine
REAL64TOREAL32	compiler library		predefine
REAL64TOSTRING	io library	convert	
RealIDiv	compiler library		
RealIMul	compiler library		
ReFloat	maths utility library		
ReFloat64	maths utility library		
Reinitialise	system library	xlink	
REM	language keyword		
RESULT	language keyword		
RETYPES	language keyword		
ROTATELEFT	compiler predefine		
ROTATERIGHT	compiler predefine		
ROUND	language keyword		
ROUNDSN	compiler predefine		T4, T425, TB
SC	compiler directive		
SCALEB	compiler library		
search.match	io library	string	
search.no.match	io library	string	
SEQ	language keyword		
SHIFTLEFT	compiler predefine		
SHIFTRIGHT	compiler predefine		
SHIFTRIGHT64	maths utility library		
SIN	maths library	snglmath	also tbmaths
SINH	maths library	snglmath	also tbmaths
SIZE	language keyword		
SKIP	language keyword		
so.ask	io library	hostio	
so.buffer	io library	hostio	
so.close	io library	hostio	
so.commandline	io library	hostio	
so.core	io library	hostio	
so.date.to.ascii	io library	hostio	
so.eof	io library	hostio	

Name	Class	Library	Notes
so.exit	io library	hostio	
so.ferror	io library	hostio	
so.flush	io library	hostio	
so.fwrite.char	io library	hostio	
so.fwrite.hex.int	io library	hostio	
so.fwrite.hex.int64	io library	hostio	
so.fwrite.int	io library	hostio	
so.fwrite.int64	io library	hostio	
so.fwrite.nl	io library	hostio	
so.fwrite.real32	io library	hostio	
so.fwrite.real64	io library	hostio	
so.fwrite.string	io library	hostio	
so.fwrite.string.nl	io library	hostio	
so.getenv	io library	hostio	
so.getkey	io library	hostio	
so.gets	io library	hostio	
so.keystream.from.file	io library	streamio	
so.keystream.from.kbd	io library	streamio	
so.keystream.from.stdin	io library	streamio	
so.multiplexor	io library	hostio	
so.open	io library	hostio	
so.open.temp	io library	hostio	
so.overlapped.buffer	io library	hostio	
so.overlapped.multiplexor	io library	hostio	
so.parse.command.line	io library	hostio	
so.pollkey	io library	hostio	
so.popen.read	io library	hostio	
so.puts	io library	hostio	
so.read	io library	hostio	
so.read.echo.any.int	io library	hostio	
so.read.echo.hex.int	io library	hostio	
so.read.echo.hex.int64	io library	hostio	
so.read.echo.int	io library	hostio	
so.read.echo.int64	io library	hostio	
so.read.echo.line	io library	hostio	
so.read.echo.real32	io library	hostio	
so.read.echo.real64	io library	hostio	
so.read.line	io library	hostio	

Name	Class	Library	Notes
so.remove	io library	hostio	
so.rename	io library	hostio	
so.scrstream.to.ANSI	io library	streamio	
so.scrstream.to.stdout	io library	streamio	
so.scrstream.to.TVI920	io library	streamio	
so.system	io library	hostio	
so.tell	io library	hostio	
so.test.exists	io library	hostio	
so.time	io library	hostio	
so.time.to.ascii	io library	hostio	
so.time.to.date	io library	hostio	
so.today.ascii	io library	hostio	
so.today.date	io library	hostio	
so.version	io library	hostio	
so.write	io library	hostio	
so.write.char	io library	hostio	
so.write.hex.int	io library	hostio	
so.write.hex.int64	io library	hostio	
so.write.int	io library	hostio	
so.write.int64	io library	hostio	
so.write.nl	io library	hostio	
so.write.real32	io library	hostio	
so.write.real64	io library	hostio	
so.write.string	io library	hostio	
so.write.string.nl	io library	hostio	
SQRT	compiler library		predefine
ss.beep	io library	streamio	
ss.B00x.terminal.driver	io library	process	
ss.clear.eol	io library	streamio	
ss.clear.eos	io library	streamio	
ss.del.line	io library	streamio	
ss.delete.chl	io library	streamio	
ss.delete.chr	io library	streamio	
ss.down	io library	streamio	
ss.goto.xy	io library	streamio	
ss.ins.line	io library	streamio	
ss.insert.char	io library	streamio	
ss.left	io library	streamio	
ss.right	io library	streamio	

Name	Class	Library	Notes
ss.scrstream.copy	io library	streamio	
ss.scrstream.fan.out	io library	streamio	
ss.scrstream.from.array	io library	streamio	
ss.scrstream.sink	io library	streamio	
ss.scrstream.to.array	io library	streamio	
ss.scrstream.to.file	io library	streamio	
ss.seek	io library	hostio	
ss.up	io library	streamio	
ss.write.char	io library	streamio	
ss.write.endstream	io library	streamio	
ss.write.hex.int	io library	streamio	
ss.write.hex.int64	io library	streamio	
ss.write.int	io library	streamio	
ss.write.int64	io library	streamio	
ss.write.nl	io library	streamio	
ss.write.real32	io library	streamio	
ss.write.real64	io library	streamio	
ss.write.string	io library	streamio	
ss.write.text.line	io library	streamio	
STOP	language keyword		
str.shift	io library	string	
string.pos	io library	string	
STRINGTOBOOL	io library	convert	
STRINGTOHEX	io library	convert	
STRINGTOHEX16	io library	convert	
STRINGTOHEX32	io library	convert	
STRINGTOHEX64	io library	convert	
STRINGTOINT16	io library	convert	
STRINGTOINT32	io library	convert	
STRINGTOINT64	io library	convert	
STRINGTOINT	io library	convert	
STRINGTOREAL32	io library	convert	
STRINGTOREAL64	io library	convert	
T2	compiler keyword		
T212	compiler keyword		
T222	compiler keyword		
T4	compiler keyword		
T414	compiler keyword		
T425	compiler keyword		

Name	Class	Library	Notes
T8	compiler keyword		
T800	compiler keyword		
TA	compiler keyword		
TAN	maths library	snglmath	also tbmaths
TANH	maths library	snglmath	also tbmaths
TB	compiler keyword		
TC	compiler keyword		
TIMER	language keyword		
TIMES	language keyword		
to.lower.case	io library	string	
to.upper.case	io library	string	
TRUE	language keyword		
TRUNC	language keyword		
UNPACKSN	compiler predefine		T4, T425, TB
USE	compiler directive		
VAL	language keyword		
VALOF	language keyword		
VECSPACE	compiler keyword		
WHILE	language keyword		
WORKSPACE	compiler keyword		

B Transputer instruction support

This appendix contains the list of transputer instructions supported by the toolset restricted code insertion facility, and gives the mnemonic for each instruction. All the instructions listed can be inserted into occam programs using the GUY construct.

These instructions are available when the compiler is targetted to an IMS T212, T222, M212, T414, T425 or T800, unless otherwise indicated. Instructions that are only supported when the compiler is targetted to the IMS T800, T425 and TC, are given in separate sections. For the full instruction set the reader is referred to the *'Transputer instruction set: a compiler writer's guide'*.

B.1 Direct functions

ADC	Add constant
AJW	Adjust workspace
CALL	Call
CJ	Conditional jump
EQC	Equals constant
J	Jump relative
LDC	Load constant
LDL	Load local
LDLP	Load local pointer
LDNL	Load non local
LDNLP	Load non local pointer
NFIX	Negative prefix
OPR	Operate
PFIX	Prefix
STL	Store local
STNL	Store non local

B.2 Short indirect functions

ADD	Add
BSUB	Byte subscript
DIFF	Difference
GT	Greater than
LB	Load byte
PROD	Product
REV	Reverse
SUB	Subtract
WSUB	Word subscript

B.3 Long indirect functions

AND	And
BCNT	Byte count
CCNT1	Check count from 1
CFLERR	Check single length floating point infinity or NaN (T414, T425 and TB only)
CSNGL	Check single
CSUB0	Check subscript from 0
CWORD	Check word
DIV	Divide
FMUL	Fractional multiply (T414, T425, T800, TA, TB and TC only)
LADD	Long add
LDIFF	Long difference
LDINF	Load single length infinity (T414, T425 and TB only)
LDIV	Long divide
LDPI	Load pointer to instruction
LDPRI	Load current priority
LDTIMER	Load timer
LMUL	Long multiply
LSHL	Long shift left
LSHR	Long shift right
LSUB	Long subtract
LSUM	Long sum
MINT	Minimum integer
MOVE	Move message
MUL	Multiply

NORM	Normalise
NOT	Not
POSTNORMSN	Post-normalise correction of single length floating point number (T414, T425 and TB only)
OR	Or
REM	Remainder
ROUNDSN	Round single length floating point number (T414, T425 and TB only)
SB	Store byte
SETERR	Set error
SHL	Shift left
SHR	Shift right
STTIMER	Store timer
SUM	Sum
TESTERR	Test error false and clear
TESTHALTERR	Test halt-on-error
TESTPRANAL	Test processor analysing
UNPACKSN	Unpack single length floating point number (T414, T425 and TB only)
WCNT	Word count
XDBLE	Extend to double
XOR	Exclusive or
XWORD	Extend to word

B.4 Additional instructions for IMS T425, T800 and TC

The following instructions are supported by the code insertion facility for IMS T425, T800 and TC processors.

BITCNT	Count bits set in word
BITREVNBITS	Reverse n bits in word (where $1 \leq n \leq 32$)
BITREVWORD	Reverse all bits in word
CRCBYTE	Calculate CRC on byte
CRCWORD	Calculate CRC on word
DUP	Duplicate top of stack
MOVE2DALL	Two-dimensional block copy
MOVE2DINIT	Initialise data for two-dimensional block move
MOVE2DNONZERO	Two-dimensional block copy non zero bytes
MOVE2DZERO	Two-dimensional block copy zero bytes
WSUBDB	Form double word subscript

B.5 Additional instructions for IMS T800

The following IMS T800 instructions are supported by the code insertion facility.

FPADD	Floating point add
FPB32TOR64	Bit32 to real64
FPCHKERR	Check floating error
FPDIV	Floating point divide
FPDUP	Floating duplicate
FPEQ	Floating point equality
FPGT	Floating point greater than
FPI32TOR32	Int32 to real32
FPI32TOR64	Int32 to real64
FPINT	Round to floating integer
FPLDNLADDDB	Floating load non local and add double
FPLDNLADDSN	Floating load non local and add single
FPLDNLDB	Floating load non local double
FPLDNLDBI	Floating load non local indexed double
FPLDNLMULDB	Floating load non local and multiply double
FPLDNLMULSN	Floating load non local and multiply single
FPLDNLSN	Floating load non local single
FPLDNLSNI	Floating load non local indexed single
FPLDZERODB	Load zero double
FPLDZEROSN	Load zero single
FPMUL	Floating point multiply
FPNAN	Floating point NaN
FPNOTFINITE	Floating point finite
FPORDERED	Floating point orderability
FPREMFIRST	Floating point remainder first step
FPREMSTEP	Floating point remainder iteration step
FPREV	Floating reverse
FPRTOI32	Real to int32
FPSTNLDB	Floating store non local double
FPSTNLI32	Store non local int32
FPSTNLSN	Floating store non local single
FPSUB	Floating point subtract
FPTESTERR	Test floating error false and clear
FPUABS	Floating point absolute
FPUCHKI32	Check in range of type int32
FPUCHKI64	Check in range of type int64

FPUCLRERR	Clear floating point error
FPUDIVBY2	Divide by 2.0
FPUEXPDEC32	Divide by 2**32
FPUEXPINC32	Multiply by 2**32
FPUMULBY2	Multiply by 2.0
FPUNOROUND	Real64 to real32 without rounding
FPUR32TOR64	Real32 to real64
FPUR64ROR32	Real64 to real32
FPURM	Set rounding mode to round minus
FPURN	Set rounding mode to 'nearest'
FPURP	Set rounding mode to round positive
FPURZ	Set rounding mode to round zero
FPUSETERR	Set floating point error
FPUSQRTFIRST	Floating point square root first step
FPUSQRTLAST	Floating point square root end
FPUSQRTSTEP	Floating point square root step

C Constants

This appendix lists the constants provided with the occam libraries. The constants are supplied in source files on the library search path and are given the extension `.inc` (for 'include').

There are four separate files containing toolset constants, as follows:

File	Contents
`hostio.inc`	Hostio tag values and protocols
`streamio.inc`	Streamio tag values
`mathvals.inc`	Mathematical constants
`linkaddr.inc`	Transputer link addresses

To use any of these files in a program, incorporate the file into the source using the `#INCLUDE` directive as follows:

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

Constants must be declared before they are used in a program or library.

C.1 Hostio constants

```
-- SP protocol
PROTOCOL SP IS INT16::[]BYTE :

-- Command tags
-- values up to 127 are reserved for use by INMOS
-- File command tags
VAL sp.open.tag   IS 10(BYTE) :
VAL sp.close.tag  IS 11(BYTE) :
VAL sp.read.tag   IS 12(BYTE) :
VAL sp.write.tag  IS 13(BYTE) :
VAL sp.gets.tag   IS 14(BYTE) :
VAL sp.puts.tag   IS 15(BYTE) :
VAL sp.flush.tag  IS 16(BYTE) :
VAL sp.seek.tag   IS 17(BYTE) :
VAL sp.tell.tag   IS 18(BYTE) :
VAL sp.eof.tag    IS 19(BYTE) :
VAL sp.ferror.tag IS 20(BYTE) :
VAL sp.remove.tag IS 21(BYTE) :
VAL sp.rename.tag IS 22(BYTE) :

-- Host command tags
VAL sp.getkey.tag  IS 30(BYTE) :
VAL sp.pollkey.tag IS 31(BYTE) :
VAL sp.getenv.tag  IS 32(BYTE) :
VAL sp.time.tag    IS 33(BYTE) :
```

```
VAL sp.system.tag  IS 34(BYTE) :
VAL sp.exit.tag    IS 35(BYTE) :

-- Server command tags
VAL sp.commandline.tag IS 40(BYTE) :
VAL sp.core.tag        IS 41(BYTE) :
VAL sp.version.tag     IS 42(BYTE) :

-- OS specific command tags
-- These OS specific tags will be followed by
-- another tag indicating
-- which OS specific function is required

VAL sp.DOS.tag    IS 50(BYTE) :
VAL sp.HELIOS.tag IS 51(BYTE) :
VAL sp.VMS.tag    IS 52(BYTE) :
VAL sp.SUNOS.tag  IS 53(BYTE) :

-- Command tags
-- Packet and buffer Sizes
VAL sp.max.packet.size IS 512 :
-- bytes transferred, includes length & data
VAL sp.min.packet.size IS   8 :
-- bytes transferred, includes length & data

VAL sp.max.packet.data.size IS sp.max.packet.size - 2 :
-- INT16 length
VAL sp.min.packet.data.size IS sp.min.packet.size - 2 :
-- INT16 length

-- individual command maxima
VAL sp.max.openname.size    IS sp.max.packet.data.size - 5 :
-- 5 bytes extra
VAL sp.max.readbuffer.size  IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
-- ditto for gets
VAL sp.max.writebuffer.size IS sp.max.packet.data.size - 7 :
-- 7 bytes extra
-- ditto for puts
VAL sp.max.removename.size    IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
VAL sp.max.renamename.size    IS sp.max.packet.data.size - 5 :
-- 5 bytes extra
VAL sp.max.getenvname.size    IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
VAL sp.max.systemcommand.size IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
VAL sp.max.corerequest.size   IS sp.max.packet.data.size - 3 :
-- 3 bytes extra

VAL sp.max.buffer.size IS sp.max.writebuffer.size :
-- smaller of read & write

-- Packet and buffer Sizes
-- Result values         (spr.)
```

```
VAL spr.ok                IS   0(BYTE) :

VAL spr.not.implemented   IS   1(BYTE) :
VAL spr.bad.name          IS   2(BYTE) :
-- filename is null
VAL spr.bad.type          IS   3(BYTE) :
-- open file type is incorrect
VAL spr.bad.mode          IS   4(BYTE) :
-- open file mode is incorrect
VAL spr.invalid.streamid  IS   5(BYTE) :
-- never opened that streamid
VAL spr.bad.stream.use    IS   6(BYTE) :
-- reading an output file, or vice versa
VAL spr.buffer.overflow   IS   7(BYTE) :
-- buffer too small for required data
VAL spr.bad.packet.size   IS   8(BYTE) :
-- data too big or small for packet
VAL spr.bad.origin        IS   9(BYTE) :
-- seek origin is incorrect
VAL spr.notok             IS 127(BYTE) :
-- a general fail result

-- anything 128 or above is a server dependent 'failure' result
VAL spr.operation.failed IS 128(BYTE) :

-- Predefined streams     (spid.)
VAL spid.stdin  IS 0(INT32) :
VAL spid.stdout IS 1(INT32) :
VAL spid.stderr IS 2(INT32) :

-- Open types             (spt.)
VAL spt.binary IS 1(BYTE) :
VAL spt.text   IS 2(BYTE) :

-- Open modes             (spm.)
VAL spm.input           IS 1(BYTE) :
VAL spm.output          IS 2(BYTE) :
VAL spm.append          IS 3(BYTE) :
VAL spm.existing.update IS 4(BYTE) :
VAL spm.new.update      IS 5(BYTE) :
VAL spm.append.update   IS 6(BYTE) :

-- Status values          (sps.)
VAL sps.success IS  999999999(INT32) :
VAL sps.failure IS -999999999(INT32) :

-- Seek origins           (spo.)
VAL spo.start   IS 1(INT32) :
VAL spo.current IS 2(INT32) :
VAL spo.end     IS 3(INT32) :

-- Version information    (sph., spo., spb.)
-- Host types             (sph.)
-- values up to 127 are reserved for use by INMOS
VAL sph.PC    IS 1(BYTE) :
```

```
VAL sph.NECPC IS 2(BYTE) :
VAL sph.VAX   IS 3(BYTE) :
VAL sph.SUN3  IS 4(BYTE) :
VAL sph.SUN4  IS 5(BYTE) :

-- OS types                (spo.)
VAL spo.DOS    IS 1(BYTE) :
VAL spo.HELIOS IS 2(BYTE) :
VAL spo.VMS    IS 3(BYTE) :
VAL spo.SUNOS  IS 4(BYTE) :
-- values up to 127 are reserved for use by INMOS

-- Interface Board types (spb.)
-- This determines the interface between the link and the host
VAL spb.B004  IS 1(BYTE) :
VAL spb.B008  IS 2(BYTE) :
VAL spb.B010  IS 3(BYTE) :
VAL spb.B011  IS 4(BYTE) :
VAL spb.B014  IS 5(BYTE) :
VAL spb.DRX11 IS 6(BYTE) :
VAL spb.QT0   IS 7(BYTE) :
-- values up to 127 are reserved for use by INMOS

-- Command line
VAL sp.short.commandline IS BYTE 0 :
-- remove  server's own arguments
VAL sp.whole.commandline IS BYTE 1 :
-- include server's own arguments

VAL spopt.never  IS 0 :
-- values for so.parse.commandline
VAL spopt.maybe  IS 1 :
-- indicate whether an option requires a following
VAL spopt.always IS 2 :
-- parameter

-- Time string and date lengths
VAL so.time.string.len IS 19 :
-- enough for "HH:MM:SS DD/MM/YYYY"
VAL so.date.len        IS  6 :
-- enough for DDMMYY (as integers)

-- Temp filename length
VAL so.temp.filename.length IS 6 :
-- six chars will work on anything!
```

C.2 Streamio constants

```
VAL st.max.string.size IS 256 :
VAL ft.terminated   IS  -8 : -- used to terminate a keystream
VAL ft.number.error IS -11 :

PROTOCOL KS IS INT:
```

```
PROTOCOL SS
  CASE
    st.reset
    st.up
    st.down
    st.left
    st.right
    st.goto; INT32; INT32
    st.ins.char; BYTE
    st.del.char
    st.out.string; INT32::[]BYTE
    st.clear.eol
    st.clear.eos
    st.ins.line
    st.del.line
    st.beep
    st.spare
    st.terminate
    st.help
    st.initialise
    st.out.byte; BYTE
    st.out.int; INT32
    st.key.raw
    st.key.cooked
    st.release
    st.claim
    st.endstream
:
```

C.3 Maths constants

```
-- REAL32 Constants

VAL REAL32 INFINITY RETYPES #7F800000(INT32) :
VAL REAL32 MINREAL  RETYPES #00000001(INT32) :
VAL REAL32 MAXREAL  RETYPES #7F7FFFFF(INT32) :
-- 3.40282347 E+38
VAL REAL32 E        RETYPES #402DF854(INT32) :
-- 2.71828174 E+00
VAL REAL32 PI       RETYPES #40490FDB(INT32) :
-- 3.14159274 E+00
VAL REAL32 LOGE2    RETYPES #3F317218(INT32) :
-- 6.93147182 E-01
VAL REAL32 LOG10E   RETYPES #3EDE5BD9(INT32) :
-- 4.34294492 E-01
VAL REAL32 ROOT2    RETYPES #3FB504F3(INT32) :
-- 1.41421354 E+00
VAL LOGEPI IS 1.1447298858(REAL32) :
VAL RADIAN IS 57.295779513(REAL32) :
VAL DEGREE IS 1.74532925199E-2(REAL32) :
VAL GAMMA  IS 0.5772156649(REAL32) :
```

```
-- REAL64 Constants

VAL REAL64 DINFINITY RETYPES #7FF0000000000000(INT64):
VAL REAL64 DMINREAL  RETYPES #0000000000000001(INT64):
VAL REAL64 DMAXREAL  RETYPES #7FEFFFFFFFFFFFFF(INT64):
-- 1.7976931348623157E+308
VAL REAL64 DE      RETYPES #4005BF0A8B145769(INT64) :
-- 2.7182818284590451E+000
VAL REAL64 DPI     RETYPES #400921FB54442D18(INT64) :
-- 3.1415926535897931E+000
VAL REAL64 DLOGE2 RETYPES #3FE62E42FEFA39EF(INT64) :
-- 6.9314718055994529E-001
VAL REAL64 DLOG10E RETYPES #3FDBCB7B1526E50E(INT64) :
-- 4.3429448190325182E-001
VAL REAL64 DROOT2 RETYPES #3FF6A09E667F3BCD(INT64) :
-- 1.4142135623730951E+000
VAL DLOGEPI IS 1.1447298858494001741(REAL64) :
VAL DRADIAN IS 57.295779513082320877(REAL64) :
VAL DDEGREE IS 1.7453292519943295769E-2(REAL64) :
VAL DGAMMA  IS 0.57721566490153286061(REAL64) :
```

C.4 Transputer link addresses

```
VAL link0.in  IS 4:
VAL link0.out IS 0:

VAL link1.in  IS 5:
VAL link1.out IS 1:

VAL link2.in  IS 6:
VAL link2.out IS 2:

VAL link3.in  IS 7:
VAL link3.out IS 3:

VAL event.in  IS 8:
```

D ITERM

D.1 Introduction

This appendix describes the format of ITERM files; it is included for people who need to write their own ITERM because they are using terminals that are not supported by the standard ITERM file supplied with the toolset.

ITERMs are ASCII text files that describe the control sequences required to drive terminals. Screen oriented applications that use ITERM files are terminal independent.

ITERM files are similar in function to the UNIX *termcap* database and describe input from, as well as output to, the terminal. They allow applications that use function keys to be terminal independent and configurable.

Within the toolset, the ITERM file is only used by the debugger tool **idebug** and the T414 simulator tool **isim**.

D.2 The structure of an ITERM file

An ITERM file consists of three sections. These are the *host*, *screen* and *keyboard* sections. Sections are introduced by a line beginning with the section letters '**H**', '**S**' or '**K**'. Case is unimportant and the rest of the line is ignored. Sections consist of a number of lines beginning with a digit. A section is terminated by a line beginning with the letter '**E**'. The *host* section must appear first; other sections may appear in any order in the file. Sections must be separated by at least one blank line.

The syntax of the lines that make up the body of a section is best described in an example:

```
3:34,56,23,7.    comments
```

Each line starts with the index number followed by a colon and a list of numbers separated by commas. Each line is terminated by a full stop ('.') and anything following it is treated as a comment. Spaces are not allowed in the data string and an entry cannot be split across more than one line.

Comment lines, beginning with the character '#', may be placed anywhere in an ITERM file. Extra blank lines in the file are ignored.

The index numbers in each section correspond to an agreed meaning for the data. In the following sections the meaning of the data in each of the three sections is described in detail.

D.3 The host definitions

D.3.1 ITERM version

This item identifies an ITERM file by version. It provides some protection against incompatible future upgrades.

e.g. `1:2.`

D.3.2 Screen size

This item allows applications to find out the size of the terminal at startup time. The data items are the number of columns and rows, in that order, available on the current terminal.

e.g. `2:80,25.`

Screen locations should be numbered from 0, 0 by the application. Terminals which use addressing from 1, 1 can be compensated for in the definition of goto X, Y.

D.4 The screen definitions

The lists of values in the screen section represent control codes that perform certain operations; the data values are ASCII codes to send to the display device.

ITERM version 2 defines the indices given in table D.1. These definitions are used in the example ITERM file; for a complete listing of the file see section D.7.

For example, an entry like: '`8:27,91,75.`' indicates that an application should output the ASCII sequence '`ESC [ K`' to the terminal output stream to clear to end of line.

Index	Screen operation	Index	Screen operation
1	cursor up	10	insert line
2	cursor down	11	delete line
3	cursor left	12	ring bell
4	cursor right	13	home and clear screen
5	goto x y	20	enhance on
8	clear to end of line	21	enhance off
9	clear to end of screen		

Table D.1 ITERM screen operations

D.4.1 Goto X Y processing

The entry for 5, 'goto X Y', requires further interpretation by the application. A typical entry for 'goto X Y' might be:

```
5:27,-11,32,-21,32
```

The negative numbers relate to the arguments required for X and Y.

..., -ab, nn, ...

where: *a* is the argument number (i.e. 1 for X, 2 for Y).

b controls the data output format.
If *b*=1 output is an ASCII byte (e.g. 33 is output as **!**).
If *b*=2 output is an ASCII number (e.g. 33 is output as **3 3**).

nn is added to the argument before output.

As a complete example, consider the following ITERM entry in the screen section:

```
5:27,91,-22,1,59,-12,1,72. ansi cursor control
```

This would instruct an application wishing to move the terminal cursor to X=14, Y=8 (relative to 0,0) to output the following bytes to the screen:

```
Bytes in decimal: 27   91   57   59   49   53   72
Bytes in ASCII:   ESC  [    9    ;    1    5    H
```

D.5 The keyboard definitions

Each index represents a single keyboard operation. The data specified after each index defines the keystroke associated with that operation. Multiple entries

for the same index indicate alternative keystrokes for the operation.

ITERM version 2 defines the indices given in table D.2. These definitions are used in the example ITERM file; for a complete listing of the file see section D.7.

Index	Function	Index	Function
1	return	36	get address/ code address
2	delete	39	run/goto line
6	up	40	backtrace
7	down	41	inspect
8	left	42	channel
9	right	43	top/step source
14	sol	44	retrace
15	eol	45	relocate/ walk source
18	line up	46	information/ breakpoint
19	line down	47	search
20	page up	48	links/go
21	page down	49	monitor
26	enter file	50	word left
27	exit file	51	wordright
28	refresh	55	top of file
29	change file	56	end of file
31	finish/ Monitor page		
33	info		
34	help		

Table D.2 ITERM key operations

D.6 Setting up the ITERM environment variable

To use an ITERM the application has to find and read the file. An environment variable (or logical name on VMS) called '`ITERM`' should be set up with the pathname of the file as its value. For example, under MS-DOS the command would be:

```
C:\> set ITERM=C:\ITOOLS\TOOLS\IBMPC.ITM
```

Under UNIX you would set an environment variable. For example, the command for `csh` users might be:

```
%  setenv ITERM ~/.iterm
```

Under VMS you would define a logical name. For example:

```
$ DEFINE ITERM SYS$LOGIN:VT220.ITM
```

For more details see the Delivery Manual.

D.7 An example ITERM

This is the toolset ITERM file for the IBM PC using the ANSI screen driver.

```
#  ------------------------------------------------
#
#     iterm for ibm-pc (with ansi.sys)
#     support for d705 debugger and simulator
#
#     V1.1 - 16 March 89          (j3)
#
#  ------------------------------------------------

host section
1:2.                              version
2:80,25.                          screen size
end of host section

screen section
#                                 DEBUGGER        SIMULATOR
1:27,91,49,65.                                    cursor up
2:27,91,49,66.                                    cursor down
3:27,91,49,68.                    cursor left     cursor left
4:27,91,49,67.                                    cursor right
5:27,91,-22,1,59,-12,1,72.        goto x y        goto x y
8:27,91,75.                       clear to eol    clear to eol
9:27,91,50,74.                    clear to eos    clear to eos
#10          ansi.sys does        insert line     insert line
#11          not have these       delete line     delete line
12:7.                             bell            bell
13:27,91,74.                      clear screen    clear screen
end of screen section


keyboard section
#               IBM-PC KEY        DEBUGGER        SIMULATOR
2:127.        # BACKSPACE         del char
6:0,72.       # UP                cursor up       cursor up
7:0,80.       # DOWN              cursor down     cursor down
8:0,75.       # LEFT              cursor left     cursor left
```

```
9:0,77.      # RIGHT           cursor right   cursor right
14:0,65.     # F7              start of line  start of line
15:0,66.     # F8              end of line    end of line
18:0,67.     # F9              line up
19:0,68.     # F10             line down
20:0,112.    # ALT F9          page up        page up
21:0,113.    # ALT F10         page down      page down
26:0,71.     # KEYPAD 7        enter file
27:0,73.     # KEYPAD 9        exit file
28:27.       # ESC             refresh        refresh
29:0,85.     # SHIFT F2        change file
31:0,117.    # CTRL KEYPAD 1   finish         monitor page
33:0,60.     # F2              help
34:0,59.     # F1              help           help
36:0,63.     # F5              get address    code address
39:0,64.     # F6              goto line
40:0,129.    # ALT 0           backtrace      backtrace
41:0,120.    # ALT 1           inspect        inspect
42:0,121.    # ALT 2           channel        channel
43:0,122.    # ALT 3           top            step source
44:0,123.    # ALT 4           retrace        retrace
45:0,124.    # ALT 5           relocate       walk source
46:0,125.    # ALT 6           info           breakpoint
47:0,126.    # ALT 7           search         search
48:0,127.    # ALT 8           links          go
49:0,128.    # ALT 9           monitor        monitor
50:0,90.     # SHIFT F7        word left
51:0,91.     # SHIFT F8        word right
55:0,92.     # SHIFT F9        top of file
56:0,93.     # SHIFT F10       end of file
end of keyboard section

# eof
```

E Executable file format

This appendix describes the format of the executable image files that are produced by the bootstrap tool. These are:

- Bootable files which contain bootstrap code in addition to program code and which can be loaded directly onto a transputer which boots from link. These files have a `.bxx` file extension.
- Non-bootable files which contain only program code and can be used for dynamic loading using the **KERNEL.RUN** program or for programs to be loaded by code booted independently from ROM. These files have a `.rxx` file extension.

E.1 Bootable files

The bootstrap tool can only be used to produce bootable files for programs that are intended to run on a *single* processor.

A bootable file consists of three parts:

1 The *primary loader*. This is used to initialise the transputer and call in the secondary loader.

  The length of the primary loader is determined by the first byte in the file (which is unsigned), and cannot exceed 255 bytes. The bytes following the length byte will be the primary loader, up to the length specified.

2 The *secondary loader*. This is used to load the program and set up the parameters used to call the program. These parameters are standard for all bootable programs and cannot be altered; they are fixed into the secondary loader's code.

  For 32 bit transputers the length of the secondary loader is given by the first four bytes (which are signed) after the end of the primary loader. This 32 bit integer is represented in the same way as an occam **INT32** type, with the least significant byte first.

  For 16 bit transputers the length of the secondary loader is given by the first two bytes (which are signed) after the end of the primary loader. This 16 bit integer is represented in the same way as an occam **INT16** type, with the least significant byte first. This means that on 16 bit transputers the secondary loader cannot exceed 32K bytes in length.

  The block of bytes following the length bytes is the secondary loader, up

to the length specified.

3 A series of 32 bit integers and **BYTE** blocks that represent program parameters, followed by the program code. The 32 bit integers are represented in the same way as an occam **INT32** type, with the least significant byte first. The integers provide data about the program which is used by the secondary loader to set up the parameters for the call of the program. The last word before the program block gives the size of the program in bytes and is followed by the program code.

The *Interface descriptor* and *Compiler id* are included for compatibility with other software such as the IMS D700D Transputer Development System (TDS). In files created by the toolset they are not used and are represented by a pair of 32-bit integers with the value zero.

The sequence of data and code blocks in a bootable file is summarised in the following table.

Type	Value	Unit
BYTE	Primary loader code size	bytes
[]BYTE	Primary loader code block	
INT32	Secondary loader code size	bytes
or **INT16**	(for 16 bit transputers)	
[]BYTE	Secondary loader code block	
INT32	Interface descriptor size	bytes
[]BYTE	Interface descriptor	
INT32	Compiler id size	bytes
[]BYTE	Compiler id	
INT32	Target processor type	
INT32	File format version	
INT32	Program scalar workspace requirement	words
INT32	Program vector workspace requirement	words
INT32	C/FORTRAN/Pascal stack requirement	words
INT32	Program entry point offset	bytes
INT32	Program code size	bytes
[]BYTE	Program code block	

The meanings of the **INT32** parameters that precede the program block are described below.

Target This value indicates the type of transputer for which the program was

compiled. The values it may take are given in the following table.

Value	Meaning
2	T212, T222 or M212
4	T414
8	T800
9	T425

Version This value indicates the format version number of the file. If the value is less than 10 there will be no vector workspace size or C/FORTRAN/Pascal stack size values in the file. If the value is 10 there will be a vector workspace size but no C/FORTRAN/Pascal stack size. If the value is 11 then there will be both vector workspace and C/FORTRAN/Pascal stack sizes in the file. The toolset always produces files with a version number of 10 or greater.

Scalar workspace This value specifies in words the size of the workspace required for the linked program's run time stack.

Vector workspace This value specifies in words the size of the workspace required for the linked program's vector (array) data. The vector workspace is used by the program for storing the arrays and data structures that are used during program execution, and is normally separate from the run-time stack. This value will only be present in the code file if the version number is 10 or 11.

Non-occam stack size This specifies the size of the separate run time stack, in words, for non-occam code, as specified by the bootstrap '**S**' option. This value will only be present in the code file if the version number is 11.

Entry point offset This value indicates the offset from the base of the code block where the program starts. The value is given in bytes.

Code size The size of the program code block.

E.2 Non-bootable files

This format is produced by the omit bootstrap ('**R**') option. The format is identical to that for bootable files, except that the primary and secondary loader parameters and code blocks are omitted.

Non-bootable files can be loaded dynamically at run-time. They can also be loaded and run on transputer boards that boot from ROM.

The sequence of data and code blocks in non-bootable files is summarised in the following table.

Type	Value	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	C/FORTRAN/Pascal stack size	words
INT32	Program entry point offset	bytes
INT32	Program code size	bytes
[]BYTE	Program code block	

F Host file server protocol

This appendix describes the protocol of the host file server **iserver**.

F.1 The host file server iserver

The host file server **iserver** is implemented in C using ANSI standard run-time libraries to facilitate porting to other machines. This provides an easy method of porting the toolset (or programs written under the toolset) to new hosts. The server can easily be extended to accomodate a new host, but at the risk of unportability.

The source of the server and of the libraries used to communicate with the server is supplied with the toolset.

F.2 The server protocol

Every communication to and from the server is a packet consisting of a counted array of bytes. The count gives the length of the message and is sent in the first two bytes of the packet as a signed 16 bit number. The structure of a server packet is illustrated in figure F.1.

This protocol has been given the name **SP**, and is defined in occam as follows:

```
PROTOCOL SP IS INT16::[]BYTE :
```

F.2.1 Packet size

There is a maximum packet size of 512 bytes and a minimum packet size of 8 bytes in the to-server direction (i.e. a minimum message length of 6 bytes). The server may take advantage of this knowledge.

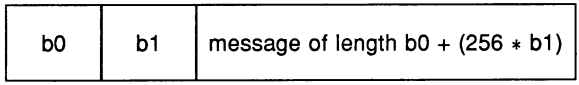


Figure F.1 **SP** protocol packet

The packet size must always be an even number of bytes. If the number of bytes is odd a dummy byte is added to the end of the packet and the packet byte count rounded up by one.

The hostio library contains routines that ensure that the size restrictions are met when sending a packet to the server (see section F.3).

F.2.2 Protocol operation

Every request sent to the server receives a reply of the same protocol, in strict sequence, and no further requests are accepted until the reply has been sent.

All integer types used by the protocol are signed and are little endian. Numbers are transmitted as sequences of bytes (2 bytes for 16 bit numbers, 4 bytes for 32 bit numbers) with the least significant byte first. Negative integers are represented in 2s complement. Strings and other variable length blocks are introduced by a 16 bit signed count.

All server calls return a result byte as the first item in the return packet. If the operation succeeds the result byte is zero and if the operation fails the result byte is non-zero. The result is one (1) in the special case where the operation fails because the function is not implemented[1]. If the result is non-zero, some or all of the return values may not be present, resulting in a smaller return packet than if the call was successful.

F.3 The server libraries

The hostio library **hostio.lib** contains all the routines provided in the toolset for communicating with the server. It contains a set of basic routines, hidden from the user, from which the more complex user visible routines are built.

A naming convention has been adopted for the server libraries. The basic library routines use the server protocol directly and map directly to server functions. These have the prefix '**sp.**'. Routines which use the basic routines and are visible to the user have the prefix '**so.**'. The '**so.**' routines documented in this manual use underlying '**sp.**' routines, and in some cases the mapping is one to one.

The source of the hostio library is provided with the toolset and serves as an example of how to use the **SP** protocol.

If you add your own libraries for server functions you are recommended to keep

[1]Result values between 2 and 127 are defined to have particular meanings by occam server libraries. Result values of 128 or above are specific to the implementation of a server.

to the naming convention.

There are two '**sp.**' library routines included to help you extend the set of available routines. These are **sp.send.packet** and **sp.receive.packet**. These are described below.

sp.send.packet

```
PROC sp.send.packet (CHAN OF SP ts,
                     VAL []BYTE packet,
                     BOOL error)
```

This procedure sends a packet on the channel **ts**, provided that it meets the requirements for a **SP** protocol packet. If the requirements are not met then the packet is not sent and **error** is set to **FALSE**.

sp.receive.packet

```
PROC sp.receive.packet (CHAN OF SP fs,
                        INT16 length,
                        []BYTE packet,
                        BOOL error)
```

This procedure receives a packet on the channel **fs**. The value **error** is set to **FALSE** if the packet exceeds the maximum packet size.

F.4 Porting the server

In order to port the **iserver** to a new machine you must have a C compiler for that machine with ANSI standard libraries. A Makefile that can assist with porting to a new machine is supplied on the toolset 'source' subdirectory.

The hostio library expects all the functions described below to be provided by **iserver**.

F.5 Defined protocol

The functions provided by the **iserver** are split into three groups:

1 File commands, for interacting with files

2 Host commands, for interacting with the host

3 Server commands, for interacting with the server.

In the descriptions that follow, the arguments and results of server calls are listed in the order that they appear in the data part of the packet. The size of a packet is the aggregated size of all the items in the packet, rounded up to an even number of bytes. occam types are used to define data items within the packet.

F.5.1 Reserved values

INMOS reserves the following values for its own use:

- Function tags in the range 0 to 127 inclusive.
- Result values in the range 0 to 127 inclusive.
- Stream identifiers 0, 1 and 2.

Some commands may return particular values, which may be reserved. The range of reserved values is given with each command as appropriate.

F.5.2 File commands

Open files are identified with 32 bit descriptors. There are three predefined open files:

0 – standard input
1 – standard output
2 – standard error

If one of these is closed then it may not be reopened.

Fopen – Open a file

```
Synopsis:    StreamId = Fopen( Name, Type, Mode )

To server:    BYTE            Tag = 10
              INT16::[]BYTE   Name
              BYTE            Type = 1 or 2
              BYTE            Mode = 1...6

From server:  BYTE            Result
              INT32           StreamId
```

Fopen opens the file **Name** and, if successful, returns a stream identifier **StreamId**.

Type can take one of two possible values:

1 Binary. The file will contain raw binary bytes.

2 Text. The file will be stored as text records. Text files are host-specified.

Mode can have 6 possible values:

1 Open an existing file for input.

2 Create a new file, or truncate an existing one, for output.

3 Create a new file, or append to an existing one, for output.

4 Open an existing file for update (both reading and writing), starting at the beginning of the file.

5 Create a new file, or truncate an existing one, for update.

6 Create a new file, of append to an existing one, for update.

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions, however. An output operation may not follow an input operation without an intervening Fseek, Ftell or Fflush operation.

The number of streams that may be open at one time is host-specified, but will not be less than eight (including the three predefines).

Fclose – Close a file

```
Synopsis:      Fclose( StreamId )

To server:     BYTE            Tag = 11
               INT32           StreamId

From server:   BYTE            Result
```

Fclose closes a stream **StreamId** which should be open for input or output. Fclose flushes any unwritten data and discards any unread buffered input before closing the stream.

Fread – Read a block of data

```
Synopsis:     Data = Fread( StreamId, Count )

To server:    BYTE           Tag = 12
              INT32          StreamId
              INT16          Count

From server:  BYTE           Result
              INT16::[]BYTE  Data
```

Fread reads **Count** bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If **Count** is less than one then no input is performed. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

Result is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

Fwrite – Write a block of data

```
Synopsis:     Written = Fwrite( StreamId, Data )

To server:    BYTE           Tag = 13
              INT32          StreamId
              INT16::[]BYTE  Data

From server:  BYTE           Result
              INT16          Written
```

Fwrite writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of **Data** is less than zero then no output is performed. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position is undefined.

Fwrite returns the number of bytes actually output in **Written**. **Result** is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

If **StreamId** is **1** (standard output) or **2** (standard error) then the write is automatically flushed.

Fgets – Read a line

```
Synopsis:      Data = Fgets( StreamId, Count )

To server:     BYTE           Tag = 14
               INT32          StreamId
               INT16          Count

From server:   BYTE           Result
               INT16::[]BYTE  Data
```

Fgets reads a line from a stream which must be open for input. Characters are read until end of file is reached, a newline character is seen or the number of characters read is not less than **Count**.

If the input is terminated because a newline is seen then the newline sequence is *not* included in the returned array.

If end of file is encountered and nothing has been read from the stream then Fgets fails.

Fputs – Write a line

```
Synopsis:      Fputs( StreamId, String )

To server:     BYTE           Tag = 15
               INT32          StreamId
               INT16::[]BYTE  String

From server:   BYTE           Result
```

Fputs writes a line of text to a stream which must be open for output. The host-specified convention for newline will be appended to the line and output to the file. The maximum line length is host-specified.

Fflush – Flush a stream

```
Synopsis:      Fflush( StreamId )

To server:     BYTE            Tag = 16
               INT32           StreamId

From server:   BYTE            Result
```

Fflush flushes the specified stream, which should be open for output. Any internally buffered data is written to the destination device. The stream remains open.

Fseek – Set position in a file

```
Synopsis:      Fseek( StreamId, Offset, Origin )

To server:     BYTE            Tag = 17
               INT32           StreamId
               INT32           Offset
               INT32           Origin

From server:   BYTE            Result
```

Fseek sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be `Offset` characters from `Origin` which may take one of three values:

1 `Set`, the beginning of the file

2 `Current`, the current position in the file

3 End, the end of the file.

For a text stream, `Offset` must be zero or a value returned by Ftell. If the latter is used then `Origin` must be set to 1.

Ftell – Find out position in a file

```
Synopsis:      Position = Ftell( StreamId )

To server:     BYTE            Tag = 18
               INT32           StreamId

From server:   BYTE            Result
               INT32           Position
```

Ftell returns the current file position for **StreamId**.

Feof – Test for end of file

```
Synopsis:      Feof( StreamId )

To server:     BYTE            Tag = 19
               INT32           StreamId

From server:   BYTE            Result
```

Feof succeeds if the end of file indicator for **StreamId** is set.

Ferror – Get file error status

```
Synopsis:      ErrorNo, Message = Ferror(StreamId)

To server:     BYTE            Tag = 20
               INT32           StreamId

From server:   BYTE            Result
               INT32           ErrorNo
               INT16::[]BYTE   Message
```

Ferror succeeds if the error indicator for **StreamId** is set. If it is, Ferror returns a host-defined error number and a (possibly null) message corresponding to the last file error on the specified stream.

Remove – Delete a file

```
Synopsis:      Remove( Name )

To server:     BYTE            Tag = 21
               INT16::[]BYTE   Name

From server:   BYTE            Result
```

Remove deletes the named file.

Rename – Rename a file

```
Synopsis:      Rename( OldName, NewName )

To server:     BYTE            Tag = 22
               INT16::[]BYTE   OldName
               INT16::[]BYTE   NewName

From server:   BYTE            Result
```

Rename changes the name of an existing file **OldName** to **NewName**.

F.5.3 Host commands

Getkey – Get a keystroke

```
Synopsis:      Key = GetKey()

To server:     BYTE            Tag = 30

From server:   BYTE            Result
               BYTE            Key
```

GetKey gets a single character from the keyboard. The keystroke is waited on indefinitely and will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

PollKey – Test for a key

```
Synopsis:     Key = PollKey()

To server:    BYTE            Tag = 31

From server:  BYTE            Result
              BYTE            Key
```

PollKey gets a single character from the keyboard. If a keystroke is not available then PollKey returns immediately with a non-zero result. If a keystroke is available it will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

Getenv – Get environment variable

```
Synopsis:     Value = Getenv( Name )

To server:    BYTE            Tag = 32
              INT16::[]BYTE   Name

From server:  BYTE            Result
              INT16::[]BYTE   Value
```

Getenv returns a host-defined environment string for **Name**. If Name is undefined then **Result** will be non-zero.

Time – Get the time of day

```
Synopsis:     LocalTime, UTCTime = Time()

To server:    BYTE            Tag = 33

From server:  BYTE            Result
              INT32           LocalTime
              INT32           UTCTime
```

Time returns the local time and Coordinated Universal Time if it is available. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero.

System – Run a command

```
Synopsis:      Status = System( Command )

To server:     BYTE            Tag = 34
               INT16::[]BYTE   Command

From server:   BYTE            Result
               INT32           Status
```

System passes the string **Command** to the host command processor for execution. If Command is zero length then System will succeed if there is a command processor. If Command is not null then **Status** is the return value of the command, which is host-defined.

F.5.4 Server commands

Exit – Terminate the server

```
Synopsis:      Exit( Status )

To server:     BYTE            Tag = 35
               INT32           Status

From server:   BYTE            Result
```

Exit terminates the server, which exits returning **Status** to its caller.

If **Status** has the special value 999999999 then the server will terminate with a host-specific 'success' result.

If **Status** has the special value -999999999 then the server will terminate with a host-specific 'failure' result.

CommandLine – Retrieve the server command line

```
Synopsis:      String = CommandLine( All )

To server:     BYTE            Tag = 40
               BYTE            All

From server:   BYTE            Result
               INT16::[]BYTE   String
```

CommandLine returns the command line passed to the server on invocation.

If `All` is zero the returned string is the command line, with arguments that the server recognised at startup removed.

If `All` is non-zero then the string returned is the entire command vector as passed to the server on startup, including the name of the server command itself.

Core – Read peeked memory

```
Synopsis       Data = Core( Offset, Length )

To server:     BYTE            Tag = 41
               INT32           Offset
               INT16           Length

From server:   BYTE            Result
               INT16::[]BYTE   Core
```

Core returns the contents of the root transputer's memory, as peeked from the transputer when the server was invoked with the analyse option.

Core fails if `Offset` is larger than the amount of memory peeked from the transputer or if the transputer was not analysed.

If `Offset + Length` is larger than the total amount of memory that was peeked then as many bytes as are available from the given offset are returned.

Version – Find out about the server

```
Synopsis:      Id = Version()

To server:     BYTE            Tag = 42

From server:   BYTE            Result
               BYTE            Version
               BYTE            Host
               BYTE            OS
               BYTE            Board
```

Version returns four bytes containing identification information about the server and the host it is running on.

If any of the bytes has the value 0 then that information is not available.

Version identifies the server version. The byte value should be divided by ten to yield the version number.

Host identifies the host machine and can be any of the following:

1 PC

2 NEC-PC

3 VAX

4 Sun-3

5 370 Architecture

OS identifies the host environment and can be any of the following:

1 DOS

2 Helios

3 VMS

4 SunOS

5 CMS

Board identifies the interface board and can be any of the following:

1 B004

2 B008

3 B010

4 B011

5 B014

6 DRX-11

7 QT0

8 B015

9 CAT

Values of **Host**, **OS** and **Board** from 0 to 127, inclusive, are reserved for use by INMOS.

G Glossary

Alias check A program compilation check that ensures that names are unique within a given scope.

Analyse To assert a signal to a transputer forcing it to halt at the next descheduling point, to allow the state of the processor to be read. In the context of 'analysing a network', to analyse all processors in the network.

Also refers to one of the system control functions on transputers and the pin on which the function is asserted.

Backtrace Within the debugger an simulator tools, to move from a position within a procedure or function body to the call of that procedure or function.

Bootable code Self-starting program code, that can be loaded onto a transputer or transputer network down a transputer link and run. Bootable code is produced by `iboot` for single transputer programs and `iconf` for multitransputer programs.

Bootstrap A transputer program, loaded from a ROM or over a link after the transputer has been reset or analysed, which initialises the processor and loads a program for execution (which may be another loader).

Compiler library A group of occam library routines that are used by the compiler to implement extended arithmetic and transputer system operations.

Configuration The association of components of an occam program with a set of physical resources. Used in this manual to refer to the specific case of allocating occam processes to processors in a network, and channels to links between processors. The term is also used, depending on the context, to describe the act of deciding on these allocations for a program, the occam code which describes such a set of allocations, and the act of applying the configurer to a network description.

Configurer The tool which assigns processes and channels on a specified configuration of transputers. The output from the tool is a bootable program.

Deadlock A state in which one or more concurrent processes can no longer proceed because of a communication interdependency.

Error mode The compilation mode of a program that determines what happens when a program error (such as an array bounds violation) occurs. A program compiled using the toolset may be compiled in one of four error modes: HALT, STOP, UNDEFINED, or UNIVERSAL.

Error signal In the transputer, an external signal used to indicate that an error has occurred in a running program. Also refers to one of the system control functions on transputers. Error signals can be OR-ed together on transputer boards to indicate an error has occurred in one of the transputers in the network.

Extended data types occam data types **INT16**, **INT32**, **INT64**, **REAL32** and **REAL64**.

Hard channels Channels which are mapped onto links between processors in a transputer network (cf. *Soft channels*).

Host The computer which is running the toolset host file server and providing the filing system and terminal i/o.

Host file server A file server which provides access to the filing system and terminal i/o of a host operating system, which may be used when running standalone programs. The toolset host file server is distinct from that used to run the Transputer Development System (TDS).

Include file A file containing source code which is incorporated into a program using the **#INCLUDE** directive. Include files that contain only declarations by convention are given the **.inc** extension.

Library A collection of separately compiled procedures or functions, created by the toolset librarian **ilibr**, which may be shared between parts of a program or between different programs.

Library build file A file containing a list of input files for the librarian tool **ilibr**. Each file forms a separately loadable module in the library. Library build files must have the **.lbb** extension.

Library usage file A file listing the libraries and separately compiled units used by another library. Library usage files must have the **.liu** extension.

Link In the context of transputer hardware, the serial communication link between processors. Used as a verb in the context of program compilation, to collect together all the code for a program or compilation unit, resolving all references and recompiling where necessary, and place the collected code into a single file.

Linker The program or tool which links a program or compilation unit.

Loader Depending on the context, refers to the part of the host file server which loads a transputer network or to a small program which is loaded into a transputer, and which then distributes code to other transputers and loads a larger program on top of itself.

Makefile An input file for a Make program. A Makefile contains details of file dependencies and directions for rebuilding the object code. Makefiles are created for the toolset using **`imakef`**.

Network A set of transputers connected together using links as a connected graph, that is, in such a way that there is a path, via links and other transputers, from each transputer to every other transputer in the set.

Newline sequence The sequence of ASCII characters, defined within the host file server, that directs a new line to be started on the terminal display or within a file. Defined for the toolset as the sequence 'CR LF'.

Object code Intermediate code between occam source and bootable files. Object code cannot be directly loaded onto a transputer and run. The compiler and linker tools generate object code.

Peek and poke To read and write locations in a transputer's memory, by communication over a link, while the transputer is waiting for a bootstrap.

Preamble The part of a transputer loader program that initialises the state of the processor.

Priority In the transputer, the priority level at which the currently executing process is being run. The IMS T800, IMS T425, IMS T414 and the IMS T212 all support two levels of priority, known as 'high' and 'low'.

Process Self-contained, independently executable occam code.

Protocol The pattern of communications between two processes, often including communications on more than one channel. When appearing as PROTOCOL, refers to a specific communication structure on an occam channel (see the *'occam 2 Reference Manual'*).

Reset The transputer system initialisation control signal. Also refers to the pin on which the signal is asserted.

Root transputer (or Root processor) The processor in a transputer network which is physically connected to the host computer, and through which the network is loaded or analysed.

Separate compilation A self-contained part of a program may be separately compiled, so that only those parts of a program which have changed since the last compilation need to be recompiled.

Server A program running in the host computer attached to a transputer network, which provides access to the filing system and terminal i/o of the host computer. The server can also be used to load the program onto the network.

Soft channels Channels declared and used within a process running on a single transputer. (cf. *Hard channels*). Soft channels are implemented by a single word in memory.

Standard error The host system error handler. Errors directed to standard error are displayed in a host-defined way, for example, on the terminal screen. For details of how to modify standard error on the system, consult the operating system documentation.

Standard input The host system input handler. Specifies the standard input device, for example the terminal keyboard or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Standard output The host system output handler. Specifies the standard output device, for example, the terminal screen or a disk file. For details of how

to modify standard input on the system, consult the operating system documentation.

Subsystem In transputer board architecture, the combination of the Reset, Analyse and Error signals which allows the board to control another board on its subsystem port.

Target transputer The transputer on which the code is intended to run. The transputer type, or a restricted set of types defined in a transputer class, is defined when the program is compiled, using command line options.

Usage check A compilation check that ensures no variables are shared between parallel processes, and that enforces rules about the use of channels as unidirectional point-to-point connections.

Vector space The data space required for the storage of vectors (arrays) within an occam program.

Workspace The data space required by an occam process; when used in contrast to *Vector space*, refers to the data space required for scalars within the process.

H Bibliography

This appendix contains a list of some transputer-related publications which may be of interest to the reader.

H.1 INMOS publications

D Pountain and D May
A tutorial introduction to occam programming
Blackwell Scientific 1987.

INMOS
occam 2 Reference Manual
Prentice Hall 1988.

INMOS
occam
Keigaku Shuppan Publishing Company 1984
(In Japanese)

INMOS
Transputer reference manual
Prentice Hall 1988

INMOS
Transputer development system
Prentice Hall 1988

INMOS
Transputer instruction set: a compiler writer's guide
Prentice Hall 1988

H.2 INMOS technical notes

R Shepherd
Extraordinary use of transputer links
Technical note 1
72 TCH 001

P Moore
IMS B010 NEC add-in board
Technical note 8
72 TCH 008

S Ghee
IMS B004 IBM PC add-in board
Technical note 11
72 TCH 011

P Atkin
Performance maximisation
Technical note 17
72 TCH 017

N Miller
Exploring Multiple Transputer Arrays
Technical note 24
72 TCH 024

G Harriman
Notes on graphics support and performance improvements on the IMS T800
Technical note 26
72 TCH 026

R Shepherd and P Thompson
Lies, damned lies, and benchmarks
Technical note 27
72 TCH 027

R Shepherd
Security aspects of occam 2
Technical note 32
72 TCH 032

J M Wilson
Analysing transputer networks
Technical note 33
72 TCH 033

J M Wilson
Loading transputer networks
Technical note 34
72 TCH 034

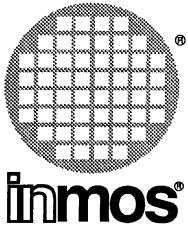
S Redfern
Implementing data structures and recursion in occam
Technical note 38
72 TCH 038

T Watson
Module motherboard architecture
Technical note 49
72 TCH 049

J Packer
Simple real-time programming with the transputer
Technical note 51
72 TCH 051

A Hamilton
Using the D705B occam toolset with non-occam applications
Technical note 55
72 TCH 055

Index



INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
U.K.
Telephone (0454) 616616
TLX 444723

INMOS Corporation
P.O. Box 16000
Colorado Springs
Colorado 80935
U.S.A.
Telephone (719) 630 4000
TLX (Easy Link) 62944936

INMOS SARL
Immeuble Monaco
7 rue Le Corbusier
SILIC 219
94518 Rungis Cedex
France
Telephone (1) 46.87.22.01
TLX 201222

INMOS Japan K.K.
4th Floor No 1 Kowa Bldg
11-41 Akasaka 1-chome
Minato-ku
Tokyo 107
Japan
Telephone 03-505-2840
TLX J29507 TEI JPN

INMOS GmbH
Danziger Strasse 2
8057 Eching
West Germany
Telephone (089) 319 10 28
TLX 522645