inmos®

occam 2 Toolset User Guide

SGS-THOMSON MICROELECTRONICS

INMOS is a member of the SGS–THOMSON Microelectronics Group

INMOS document number: 72 TDS 366 01

Contents overview

Appendices

Contents

Preface

Host versions

The documentation set which accompanies the occam 2 toolset is designed to cover all host versions of the toolset:

- IMS D7305 – IBM PC compatible running MS–DOS
- IMS D4305 – Sun 4 systems running SunOS.
- IMS D6305 – VAX systems running VMS.

About this manual

This manual is the *User Guide* to the occam 2 toolset and is divided into two parts: '*Basics*' and '*Advanced Techniques*' plus appendices. In addition some chapters are generic to other INMOS toolsets.

Differences from the previous release of the occam 2 toolset are listed immediately after this preface.

The basic section introduces the transputer and the toolset; provides an overview of the development cycle and then provides a chapter on each of the following:

- Getting started – a tutorial.
- Parallel programming using a single transputer.
- The configuration process.
- Loading programs onto a transputer network.
- Access to host services.
- Debugging programs with the toolset debugger `idebug`.

The advanced section is aimed at the more experienced user and covers the following topics:

- Advanced use of the configurer including placing code and data at specific memory locations and the software virtual through-routing mechanism.
- Mixed language programming.
- Developing programs for EPROM.
- Low level programming facilities provided by the toolset e.g. dynamic code loading.

The appendices provided in the *User Guide* include a glossary of terms and a bibliography.

Examples used in this manual

Sources for many of the examples used in this manual can be found in the `examples/manuals` subdirectory supplied with the toolset. This directory is further subdivided to group related example files. '`Readme`' files provide guidance on the content of the `examples` subdirectories, together with brief instructions about how to build the examples.

Only complete examples are provided in source form, code fragments listed in the manuals are not included.

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 366 01 occam 2 Toolset User Guide* (this manual)
- *72 TDS 367 01occam 2 Toolset Reference Manual*

  Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products, e.g. the ANSI C and FORTRAN toolsets, and the documentation reflects this – examples may be given in more than one language. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.
- *72 TDS 368 01 occam 2 Toolset Language and Libraries Reference Manual*

  Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details of extensions to the language are given in an appendix.
- *72 TDS 379 00 Performance Improvement with the INMOS Dx305 occam 2 Toolset*

  This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents, particularly from the *Language and Libraries Reference Manual.* **Note:** details of how to manipulate the software virtual through-routing mechanism are also given in the *User Guide.*
- *72 TDS 377 00 occam 2 Toolset Handbook*

  A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual.*

- *72 TDS 378 00 occam 2 Toolset Master Index*

  A separately bound master index which covers the *User Guide*, *Toolset Reference Manual*, *Language and Libraries Reference Manual* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.
- '*occam 2 Reference Manual*' published by Prentice Hall.
- '*A Tutorial Introduction to occam Programming*' published by BSP Professional Books.

FORTRAN toolset

At the time of writing the FORTRAN toolset product referred to in this document set is still under development and specific details relating to it are subject to change.

INQUEST

The INQUEST products referred to within this document are INMOS window-based debugging and profiling products, which may be bought separately and used with the toolset.

Documentation conventions

The following typographical conventions are used in this manual:

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

Basics

1 Introduction to transputers

This chapter introduces transputers and the programming models which may be adopted when designing programs for the transputer. It describes the main features of the transputer and transputer systems, and introduces the Communicating Sequential Process (CSP) model of parallel processing.

1.1 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware and external communication links. They can be connected one-to-another by their INMOS serial links in application-specific ways and may be used as building blocks for complex parallel processing networks or as powerful dedicated microprocessors.

The transputer is a complete microcomputer on a single chip. In addition to hardware support for concurrent programming and inter-processor communication it contains:

- A very fast (single cycle) on-chip memory.
- A programmable memory interface that allows external memory and memory mapped devices to be added with the minimum of supporting logic.
- System services for integrating transputer systems.
- Real time clocks.
- On the T8 series, an integral floating point unit.

Figure 1.1 shows the generalized architecture of the INMOS family of 32-bit transputers. 16-bit transputers are also available.

1.1.1 Transputer links

Links allow processes running on connected processors to exchange data and synchronize their activity. Support for link communications is implemented in hardware on each transputer chip. Communications down links operate concurrently with the processing unit and data can be transferred simultaneously on all links. Most transputers have four links except the IMS M212 and T400 transputers which have just two links.

Transputer links allow tools such as debugging programs to examine memory directly, from a remote processor. Links also provide a means of loading programs onto a network from the host down a single transputer link. Alternatively a network can be loaded via its links from a ROM on a single transputer.

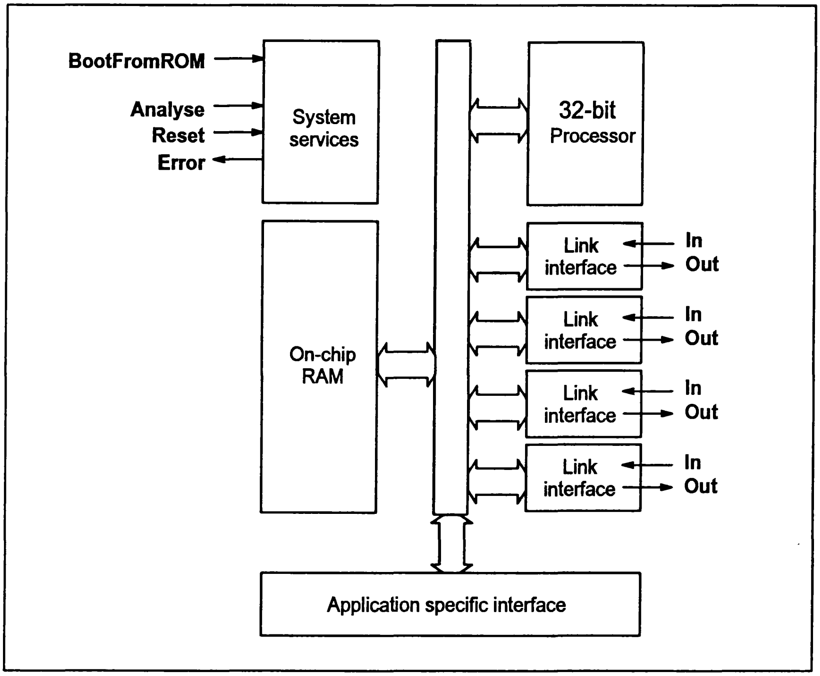


Figure 1.1 Transputer architecture

1.1.2 Process scheduling

Each transputer has a highly efficient run-time scheduler for time-sharing user application processes running on the same transputer. Within a single transputer communication between processes is supported using single words in memory. Processes waiting for input or output, or waiting for a time-slice, consume no CPU resources, and process context switching time is often less than one microsecond.

1.1.3 Real time programming

Features of the transputer provide direct hardware support for real time programming. The key features are:

- Direct and efficient implementation of parallel processes in hardware.
- Prioritization of parallel processes.
- Simple implementation of interrupt handling software.
- Easy programming of software timers, allowing close control of timing and non-busy polling.
- Placement of variables at specific addresses in memory, for accessing memory mapped devices.

Direct support for these features can be found in the current range of INMOS language toolsets, which use a common code format to facilitate code compatibility.

1.1.4 Multitransputer systems

Multitransputer systems can be built very simply using the four high speed links; only two wires are required to connect two links together. The circuitry to drive the each link is on the transputer chip.

Transputers may be connected by their INMOS links in many configurations, depending on the needs of the application. Some possible arrangements of networks of transputers are illustrated in Figure 1.2.

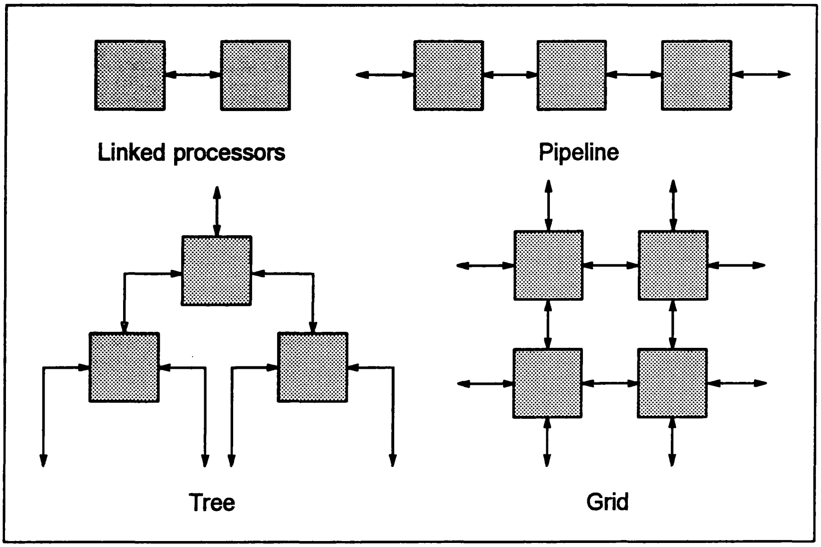


Figure 1.2 Transputer networks

1.2 Programming models

Programs developed for running on a single transputer can be designed using traditional sequential programming methods or they can be designed to exploit parallelism.

Parallelism can be designed into a program at two levels by dividing the program up into a number of independent communicating processes capable of operating in parallel. Such processes can either be run on a single transputer or on a network of transputers. Programs designed for running on a network of transputers must use the parallel processing model. See section 1.2.1.

Sequential programs can be run on a single transputer connected to a host. Such programs can exploit the transputer architecture and software support provided by INMOS toolsets and *i*q systems products, see section 1.3.

1.2.1 Parallel processing model

The abstract programming model which the transputer supports is the Communicating Sequential Process (CSP) model, based on the idea of independent parallel **processes** communicating through **channels**. Channels are one-way, point-to-point communication paths that allow processes to exchange data and synchronize their activity. (Further details can be found in '*Communicating Sequential Processes*' – C.A.R. Hoare, published by Prentice Hall International).

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

Communication between processes is synchronized. When data is passed between two processes the output process does not proceed until the input process is ready and vice versa.

Communication between software processes running on the same transputer takes place through internal channels implemented as words in memory; communication between processes running on connected processors is driven by the link interfaces and takes place through the transputer links.

1.3 Transputer products

There is a complete family of transputer devices, including: 32-bit and 16-bit processors; a link switch; and an adaptor from a parallel port to a link.

A wide range of INMOS *i*q systems transputer boards is available for specific hosts. These can be used for:

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

1.3.1 Toolset products

The INMOS compiler toolsets are complete cross-development systems for transputers. They allow transputers to be programmed sequentially and in parallel using high-level languages, making optimum use of the transputer's built-in parallel features. The combination of access to parallelism from a high level language and a set of tools for configuring and loading programs on transputer-based systems forms a powerful development system for all parallel and embedded software applications.

2 Introduction to the toolset

This chapter introduces the occam 2 toolset. It describes the main features of the toolset and provides introductions to the occam 2 compiler, the toolset linker, the configuration system, and mixed language programming. A summary of the toolset components is given at the end of the chapter.

2.1 Introduction

The occam 2 toolset is a software cross-development system for transputers, hosted on PC/MS-DOS, Sun 4/SunOS and VAX/VMS systems. It consists of an occam 2 compiler, a multilanguage linker, configuration and code collection tools, a host file server, and program development tools.

The program development tools include an interactive and post-mortem debugger, a librarian, an object code lister, a makefile generator, and EPROM programming tools. Together with the compilation system, these form an integrated support and development environment for the programming of transputers and transputer-based hardware.

2.2 Toolset features

This toolset incorporates a number of important features:

- Standard object code format generated by the compiler and linker.
- An updated occam 2 compiler with language improvements, facilitating full exploitation of a programming model designed to support parallelism.
- A configuration language which is an extension to occam and facilitates the mapping of software to hardware. The language supports:
    - Automatic placement of channels using software routing and multiplexing processes. The ability to place channels is also retained as an option.
    - Placement of code and data at specific memory addresses.
- Support for mixed language programming through the configuration system and by specific support in the compiler.

2.3 Standard object file format

The current range of INMOS toolsets generate object code in an intermediate form known as *TCOFF* (*T*ransputer *C*ommon *O*bject *F*ile *F*ormat), that can be processed by other tools in the toolset. This standard has been adopted for the development of transputer toolsets and enables modules written in different languages to be freely mixed in the same system.

2.4 occam 2 compiler

The occam 2 compiler compiles occam source code contained within standard host format text files. Any text editor that produces ASCII files can be used to create the occam source. occam source code must conform to the definition of occam 2 which is described in the *occam 2 Reference Manual*. The compiler implements a number of non-standard language extensions (see appendix A in the *occam 2 Toolset Language and Libraries Reference Manual*).

The compiler targets the current range of INMOS transputers. Code may be generated for specific processor types or for related groups (see appendix B in the *occam 2 Toolset Reference Manual*).

Code may be generated in HALT, STOP, or UNIVERSAL occam error modes. The error mode must be the same (or compatible) for all units in the compilation, and must be the same as the linker error mode.

2.4.1 Programming model

The occam programming model consists of parallel processes communicating over channels. Processes may be on the same or different processors, communicating over internal channels or transputer links.

occam 2 has been optimized for the architecture of the transputer – parallelism is expressed directly in the language. The use of a formal mathematical framework enables occam code to be extensively checked at compile time and supports formal program proving and optimization. The inherent security of occam code coupled with efficient use of the transputer's parallel features make it a powerful tool for the development of concurrent systems.

2.4.2 Language extensions

The compiler implements a number of language extensions. These are compiler-dependent and do not form part of the occam 2 definition.

Directives supported are `#INCLUDE`, `#USE`, `#COMMENT`, `#IMPORT`, `#OPTION`, and `#PRAGMA`. `#PRAGMA` supports a number of compiler-dependent functions, including foreign language code import and name translation. These are fully described in section 1.12 of the *occam 2 Toolset Reference Manual*.

Other language extensions supported by the compiler are: assembly code insertion; memory placement; and extended channel and array constructs. See appendix A of the *occam 2 Toolset Language and Libraries Reference Manual* for details.

2.4.3 occam libraries

A comprehensive set of libraries and include files are supplied with the toolset. They include the *compiler libraries* which form part of the standard support for the occam language and a set of *user libraries* for use by the applications programmer.

The compiler libraries are used internally by the compiler; they are not intended for general use by the programmer, although some routines have been made visible (see section 1.3 in the *occam 2 Toolset Language and Libraries Reference Manual*). The compiler automatically loads the correct set of routines for the selected error mode. Compiler libraries are specified to the linker by means of target-specific *linker indirect files* (see section 3.11.2).

The user libraries provide application-level support. There are libraries to support: single length, double length, and T4-optimized maths; file-based, stream-based, and DOS-specific i/o; string handling; type conversion; link error handling; CRC coding; and debugging. Constants and definitions are supplied in include files. See the *occam 2 Toolset Language and Libraries Reference Manual* for details.

2.4.4 Low level programming

Sequences of transputer instructions can be embedded in occam code using the ASM construct. This can be useful for optimizing critical sections of code, but the facility should not be over used because it reduces the compiler's opportunity to check code.

A set of procedures are provided which enable a separately compiled and linked occam procedure to be called at runtime and incorporated in a running occam program. This facility is aimed at experienced toolset users.

Full descriptions of these facilities are given in chapter 13.

2.5 Multilanguage linker

The toolset linker takes compiled code and libraries and generates a linked unit in TCOFF format. Code can be input from any compiler system which generates TCOFF code, for example, the INMOS ANSI C compiler `icc`. Linker indirect files (command scripts to the linker) may be used to specify operations to the linker; for example, as in the linker indirect files provided with the toolset for referencing the compiler libraries (see section 3.11.2).

Linker directives, which must be referenced using linker indirect files, may be used to modify the content of the linked unit. Linker directives are described in section 9.4 of the *occam 2 Toolset Reference Manual*.

2.6 Configuration system

The configurer `occonf` generates configuration information for transputer networks from a textual *configuration description* containing separate descriptions of hardware and software. Mapping of software to hardware is performed according to a mapping description written by the user, while the mapping of channels to links can be performed automatically by the configurer or be specified by the user.

The tool prepares the program for configuring on a specific arrangement of transputers by analyzing the network description file in conjunction with the configuration file, and creating a configuration data file for the code collector tool to read. The code collector then generates the program image which may be loaded onto the hardware.

The configuration language used to write the configuration description is an extension of occam. It allows software and hardware networks to be described separately and joined by an optional software-to-hardware description. The language is a simple declarative language incorporating high-level constructs such as replication and conditional statements.

2.6.1 Software routing and multiplexing

The configurer uses software routing and multiplexing software to implement channel communication over *virtual links*. This allows many *virtual channels* to use a single physical link between processors and enables processes on non-adjacent processors to communicate directly.

Software routing and multiplexing is performed automatically by the configurer and requires no intervention on the part of the programmer. Existing configuration code can be reused – virtual routing will be employed where required unless virtual routing is specifically disabled by the configurer NV option. This option effectively allows users to revert to the functionality of the D7205/D4205/D5205/D6205 toolsets.

Future INMOS transputer devices will implement virtual channel communication directly in hardware. The presence of a software virtual routing configurer in the current toolset provides some of the functionality of future processors and is intended to ease the transition to the next generation of transputer products.

2.6.2 Code and data placement

Normally, the configurer will use up the available memory accessible to a processor by allocating the various parts of the application from the lowest address upwards. However, it is sometimes necessary to specify exactly where a piece of code or data should reside. The configurer allows the user to state where the code, workspace (stack) or vectorspace of an occam program must be placed in memory.

The transputer has some very fast RAM which the application may be required to use in a special way. The configurer can also be told to avoid this area of memory so that the user has free access to it.

2.7 Mixed language programming

The use of standard TCOFF format allows compiled and linked modules from different language sources e.g. C and occam, to be mixed in the same system. Individual linked units in TCOFF format can be mixed in any combination and placed on any processor in the network.

Calling modules written in other languages is also possible. For example, occam can call C by using library routines to set up and terminate the static and heap areas. C can call occam using a 'nolink' pragma which directs the C code to be compiled without a static base parameter, or a dummy static base parameter can be declared in the occam code.

In all mixed language calls, parameters and return values passed must be of the correct type. Lists of type equivalents between C, and occam are given in chapter 11. Where character sets differ between languages, 'translate' pragmas available in the compilers can be used to create acceptable aliases.

2.8 Toolset summary

The components of the toolset are listed in Table 2.1. Descriptions of the tools can be found in Chapter 3 which also describes the main stages of program development.

Tool	Description
`icollect`	The toolset code collector. Collects linked units into a single file for loading on a transputer network. Takes as input a configuration data file or a single linked unit.
`idebug`	The toolset network debugger. Supports post-mortem and interactive debugging of transputer programs.
`idump`	The memory dumper. An auxiliary tool for use when debugging programs on the root transputer.
`iemit`	The transputer memory configuration tool. Used for evaluating and defining memory configurations for later incorporation into ROM programs.
`ieprom`	The EPROM program formatter tool. Formats transputer bootable code for input to ROM programmers.
`ilibr`	The toolset librarian. Builds libraries of compiled code.
`ilink`	The toolset linker. Resolves external references and links separately compiled units into a single file.
`ilist`	The binary lister. Disassembles and decodes object code and displays information in a readable form.
`imakef`	The Makefile generator. Generates Makefiles for input to MAKE programs.
`imap`	The map tool which gives the addresses of functions and variables used by the program.
`iserver`	The host file server. Loads programs onto transputer hardware and provides runtime access to host services.
`isim`	The T425 simulator. Simulates program execution on an IMS T425 transputer and provides simple debugging facilities.
`iskip`	The skip loader tool. Used with **`iserver`** to load programs onto external networks over the root transputer.
`oc`	The occam compiler. Compiles code for the current range of INMOS transputers.
`occonf`	The occam configurer. Reads a configuration description and produces a configuration data file for the code collector.

Table 2.1 The occam 2 toolset

3 Developing programs for the transputer

This chapter gives an overview of the program development cycle using INMOS toolsets. It briefly describes the purpose of each tool and outlines how to use them in developing, configuring, loading and running transputer programs from the host system. The chapter also provides details of command line defaults, environment variables, and host dependencies.

3.1 Introduction

This toolset is one of a range of cross-development systems designed and developed by INMOS for transputer applications. Toolsets which are available include ANSI C, occam, and FORTRAN products.

The toolsets have been designed to make program development as simple as possible. Each toolset features a particular language compiler with full library support and then uses a common set of tools for further development stages. For example, tools are included for: creating libraries, linking code, configuring software to run on transputer networks, producing the program bootable file and for loading the application onto hardware. This means that one development methodology can be used to develop programs using a number of different programming languages. Indeed one of the features of the toolsets is that they facilitate mixed language programming.

The toolset includes support for the following functions:

- building executable code;
- loading and running code;
- debugging programs;
- preparing programs for ROM;
- obtaining information about object files.

3.2 Program development using the toolsets

Programs may be developed on the user's host system before down-loading onto either a single transputer or a network of transputers to run.

Executable code is loaded onto a transputer either from ROM or from the host system via a single transputer link onto the '*root*' transputer i.e. the transputer connected to the host. Loadable code is propagated to any other transputers in the network via the interconnecting transputer links.

Creation of executable code for a transputer or transputer network takes several stages involving the use of specific tools at each stage:

1 **Software design.**

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

  Alternatively conventional sequential programs can be developed for running on a single transputer.

2 **Write the source.**

  Source code can be written using any ASCII editor available on the host system. Code can be divided between any number of source files. Source code must conform to the syntax required by the particular language compiler used. For C this is the ANSI standard; occam source code must conform to the occam 2 language definition and FORTRAN source code to FORTRAN-77 syntax.

3 **Compile the source.**

  Each source file is compiled using the appropriate language compiler to produce one or more compiled object files in TCOFF format. Each file must be compiled for the same transputer type or for a transputer class covering several compatible types. (More information about transputer types and classes is given in the appendices of the accompanying *Toolset Reference Manual*). Commonly used object code can be combined into libraries using the librarian `ilibr`.

4 **Link the compiled units.**

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

  Programs developed for the transputer may comprise one or more linked units, created from separately compiled code and library modules. Linked units are assigned to run on a single transputer or a network of transputers during configuration. A linked unit is the smallest unit of code which may be placed on a transputer.

5 **Configure the program.**

Configuration is the process of defining how the application is to be run on hardware. It is achieved by writing a configuration description, assigning linked units to specific processors and optionally connecting them by channels. By changing the configuration description it is possible to run a program on either a single transputer or on different network topologies. The description is processed by the configurer tool to produce a configuration data file. Configuration is used for both single and multiprocessor transputer programs.

The language used to write the configuration description is determined by the toolset. The C and FORTRAN toolsets provide a common configurer, `icconf` which can be used to configure programs written in C, FORTRAN-77 or occam. Using `icconf`, modules written in different languages can be mixed at configuration level, see Chapter 11. The occam toolset configurer `occonf` is designed to exploit the parallel programming model of the language and is specific to the occam toolset.

6 **Generate an executable file.**

Before a program can be run it must be made 'bootable'. This involves adding bootstrap information to make the program loadable and is achieved using the collector tool.

The configuration binary file generated by the configurer is read by the code collector `icollect` which generates a single executable file for a transputer network. The collector can generate either a file which is suitable for booting onto a transputer network via a transputer link or one for booting from ROM. The default behavior of the tool is to produce a boot-from-link executable.

Whether a boot-from-ROM executable is generated is determined by command line options specified to the configurer prior to creating the configuration binary file.

7 **Load and run the program.**

An executable boot-from-link file is loaded and run on the transputer network down a host link using `iserver`. Once loaded the code begins to execute immediately. The server tool maintains the environment that supports the program's communication with the host.

8 **Place in ROM.**

Executable boot-from-ROM files for embedded systems, are processed by the `ieprom` tool to produce an output file which is suitable for blowing into ROM. Such files may be configured to run from ROM or from RAM.

Programs to be placed in ROM are often developed first as boot-from-link, until they are error free. They are then prepared for ROM by re-submitting them to the configurer and collector, specifying different command line options, prior to using the eprom tools to format them for ROM.

Program development is supported by additional tools which provide facilities for debugging, creating object code libraries, automating the program build, and obtaining information about object files.

Figure 3.1 summarizes the main development stages.

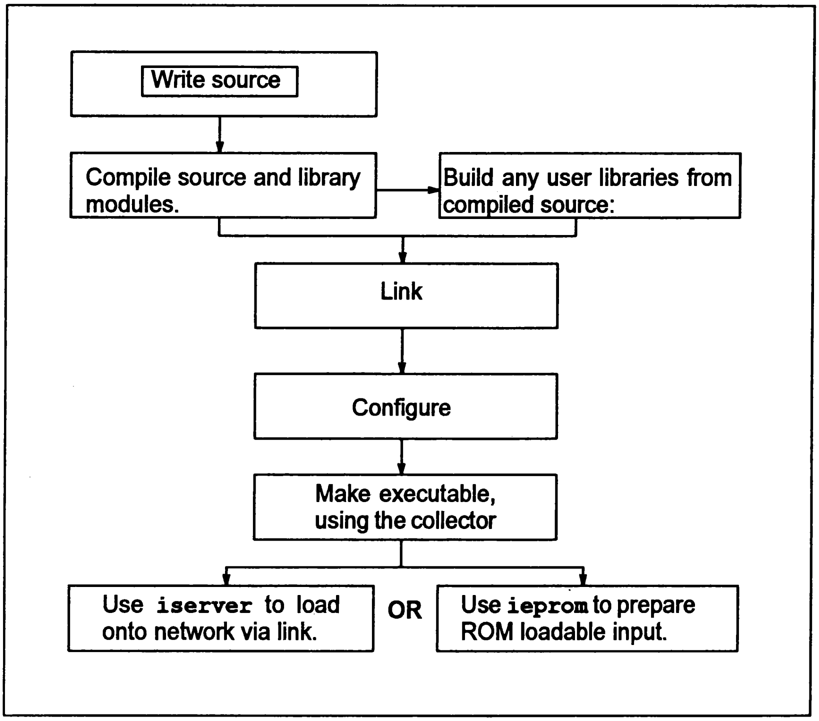

Write source
Compile source and library modules.
Build any user libraries from compiled source:
Link
Configure
Make executable, using the collector
Use iserver to load onto network via link.
OR
Use ieprom to prepare ROM loadable input.

Figure 3.1 Main development stages

3.2.1 Compatibility with previous toolset releases

For single transputer programs the configuration stage of the development process can be omitted. Instead bootable code can be generated directly from the linked unit by specifying a collector command line switch.

This mode of development is not recommended, however, and may not be supported in future toolset releases.

3.3 Compiling

INMOS compilers produce compiled code for specific processor types or for a group of related processors called a transputer class. Each compiler has the same set of options to select the target transputer; these are listed in the appendices to the accompanying *Toolset Reference Manual*. The role of transputer types and classes in compilation and program development is also described in these appendices.

The current range of INMOS compilers generate object code in an intermediate form known as *TCOFF* (*T*ransputer *C*ommon *O*bject *F*ile *F*ormat). This standard has been adopted for the development of transputer compilers and enables modules written in different languages to be freely mixed in the same system.

Supplied with each compiler is a set of language specific libraries which provide runtime support, input/output operations, mathematical functions etc. Support is also provided for language extensions, concurrent programming and software configuration of a network.

The compiler and libraries supplied with this toolset are introduced in Chapter 2. Detailed information about the compiler and libraries can be found respectively in the *Toolset Reference Manual* and the *Language and Libraries Reference Manual* supplied with this toolset.

3.4 Tools for building executable code

Three tools are used in sequence to generate the loadable file from compiled object code:

- `ilink` – the toolset linker which links separately compiled units
- `icconf` or `occonf` – the configurer tool which generates a configuration binary file.
- `icollect` – the code collector which generates a bootable file for a transputer network from the configuration data file.

The configurer works on a configuration source file written by the programmer. The output of the configurer is an information file which is processed by the collector to generate an executable or bootable file. The executable file contains all the information needed to distribute, load, and run the program on a specific network of transputers.

3.4.1 Linker – `ilink`

The toolset linker `ilink` links separately compiled modules and libraries into a single code unit, resolving external references and generating a single *linked unit*. Linked units are referenced directly from configuration descriptions to map software onto specific arrangements of transputers.

Library modules are linked in with the program by a *linker indirect file* which must be specified on the linker command line. (In the C toolset this is known as a startup file). The correct linker indirect file must be specified, depending which version of the compiler or runtime libraries is required, see section 3.11 for further details.

3.4.2 Configurer

The configurer generates configuration information for transputer networks from a textual *configuration description.* The tool prepares the application for configuring on a specific arrangement of transputers by analyzing the configuration description and creating a configuration binary file for the code collector tool to read.

Configuration descriptions are written using the transputer *configuration language* appropriate to the configurer used, see above.

3.4.3 Code collector – `icollect`

The code collector tool `icollect` takes the binary file generated by the configurer (which references the linked code) and generates a single file that can be loaded and run on a transputer network. The collector generates bootstrap and loading code. The output from the collector contains bootable code modules together with distribution information that is used by the loading code to place the correct modules on each processor.

The collector may also generate non-bootable output files which may be dynamically loaded or loaded onto ROM or RAM.

3.5 Loading and running programs

Boot-from-link code for single transputers and transputer networks is output from `icollect` and is loaded onto the transputer hardware using the host file server tool `iserver`. The `iskip` tool can be used in combination with `iserver` to load a program onto an external network, skipping the root transputer (the transputer connected to the host).

Boot-from-ROM code is processed by the eprom programming tools introduced in section 3.7.

3.5.1 Host file server – `iserver`

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

3.5.2 Skip loader – `iskip`

The skip loader `iskip` forces a program to be loaded over the root transputer (the transputer connected to the host). `iskip` is loaded prior to invoking `iserver` for loading user programs onto a transputer board and prevents the root transputer being used as part of the configured network. It continues to run as long as the user program and passes messages between the host and the network.

The tool is useful when debugging programs because it leaves the root transputer free to run the debugger. This avoids the use of `idump` to save the program image and allows the user program to run on a network that would not support the debugger e.g. because it has not enough memory.

3.6 Program development and support

Several tools are provided to assist in program development:

- `idebug` – the interactive network debugger.
- `idump` – the memory dump tool for use with `idebug` when debugging programs on the root transputer.
- `ilibr` – the librarian which generates libraries of compiled code.
- `ilist` – the binary lister which decodes and displays object files.
- `imakef` – the Makefile generator which creates Makefiles for use with MAKE programs.
- `imap` – the map tool which generates a memory map of the functions and variables used by the program.
- `isim` – the T425 simulator tool which enables programs to be executed in the absence of transputer hardware.

3.6.1 Network debugger – `idebug`

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

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

The debugger inserts no additional code into the program, but uses parallel processing to monitor the program and display its state. This guarantees that the code generated when debugging is disabled will always run in the same way as the final version of the program.

3.6.2 Memory dumper – `idump`

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

3.6.3 Librarian – `ilibr`

The librarian **`ilibr`** creates libraries of compiled code for use in application programs.

A library is a concatenation of compiled files called modules. The linker only links in modules that are required.

Code compiled by compatible TCOFF toolsets can be mixed in the same library.

3.6.4 Binary lister – `ilist`

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

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

3.6.5 Makefile generator – `imakef`

The Makefile generator **`imakef`** creates Makefiles for specific program compilations. Coupled with a suitable 'make' program it can automate building of executable code and greatly assist with code management and version control. **Note:** a make program is not supplied.

`imakef` constructs a dependency graph for a given object file and generates a Makefile in standard format. In order to make use of the tool a special set of file extensions for source and object files *must* be used throughout program development. **`imakef`** uses these file extensions to deduce target transputer types and other options. These extensions are described for **`imakef`** in the *Toolset Reference Manual.*

3.6.6 Memory map tool – `imap`

The memory map tool **`imap`** takes the text output from the toolset compiler, linker and collector and creates a map of the absolute addresses of the static variables for functions. The memory map is output on the display screen or redirected to a file as the user wishes.

3.6.7 T425 simulator – `isim`

The T425 simulator tool **`isim`** simulates the operation of the T425 transputer, enabling programs to be executed in the absence of transputer hardware.

Run in interactive mode it provides low level debugging features such as the inspection of variables, registers and queues, disassembly of memory, break points, and single step execution.

Batch mode operation of the simulator allows programs to be executed without entering the debugging environment.

3.7 EPROM programming

Two tools assist with the installation of programs into ROM, namely, the EPROM programmer **`ieprom`** and the memory configurer **`iemit`**.

3.7.1 EPROM programmer – `ieprom`

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

`iemit` output can also be interpreted and the appropriate bit pattern included in the ROM, see below.

3.7.2 Memory configurer – `iemit`

Some transputers have programmable memory interfaces which may be configured for a particular memory design.

The memory interface configurer **`iemit`** allows specific transputer memory configurations to be evaluated and can output a configuration file for incorporation into ROM by **`ieprom`**. The completed configuration file can be read by **`ieprom`** and interpreted for inclusion in the ROM at the correct address. The transputer can automatically read this data when it is reset and use it to configure its memory interface.

3.8 File types and extensions

The current range of INMOS toolsets use, by default, a standard set of file extensions to identify specific files such as source, compiled object, linked units and bootable files. Certain file extensions are assumed by the tools on input, and others generated by the tools on output, unless extensions are explicitly given on the command line. For example the compiler adds the extension `.tco` to the output file unless otherwise specified.

The adoption of a standard system allows file extensions to be omitted on the command line, and permits host file system utilities to be used. The system is designed to form an integrated whole and reflects the architecture of toolset compilation.

The standard set of file extensions is not mandatory and may be modified according to personal choice, unless **`imakef`** is to be used to build the makefile. `imakef` uses a special scheme to identify processor types and error modes, as described below.

The standard system has the advantage of ready defaults but may not be readily mapped onto existing development schemes. However, if it is decided to adopt a personalized scheme then it should be reasonably formal and controlled, which is especially important across development teams.

Some extensions recognized by the toolset are used for convention only and are not interpreted by the tools in any special way. For example, the `.lib` suffix for library files and the `.inc` suffix for include files are toolset programming conventions.

The main file extensions used in developing transputer programs are listed in Table 3.1. A full list of all file extensions used by the toolset with descriptions of the file types is given in the appendices to the accompanying *Toolset Reference Manual*.

Figure 3.2 illustrates the program development process in terms of the file extension defaults used by the toolsets. The extensions assumed on input and generated on output are used to represent source and target files. Figure 3.2 highlights the differences between the different language toolsets and shows how software can be developed to be loaded onto transputer hardware directly via a transputer link or held in ROM.

Extension	Description
`.btl`	Bootable code file. Created by `icollect`.
`.btr`	Executable code minus bootstrap information. Used for input to the EPROM tool. Created by `icollect`.
`.c`	C source files. Assumed by `icc`, the ANSI C compiler.
`.cfb`	Configuration data (binary) file. Created by the configurer.
`.cfs`	Configuration description (source) file, read by the C configurer `icconf`.
`.clu`	Configuration linked unit. Created by `occonf`.
`.f77`	FORTRAN source programs. Assumed by `if77`, the FORTRAN-77 compiler.
`.h`	Header files for use in C source code.
`.inc`	Include files named in `#INCLUDE` compiler directives for occam, or `#include` statements in configuration descriptions or in FORTRAN-77 statements.
`.lbb`	Library build file. Input to `ilibr`.
`.lib`	Library object file. Created by `ilibr`.
`.liu`	Library usage files. Created and used by `imakef`.
`.lku`	Linked unit. Created by `ilink`.
`.lnk`	Linker indirect file. Input to `ilink`.
`.occ`	occam source files. Assumed by `oc`, the occam 2 compiler.
`.pgm`	Configuration description (source) file, read by the occam configurer `occonf`.
`.rsc`	Dynamically loadable code file. Created by `icollect`.
`.tco`	Compiled code file. Created by all INMOS TCOFF compilers.

Table 3.1 Toolset main file extensions

3.8.1 File extensions required by `imakef`

The Makefile generator `imakef` requires a special set of file extensions to be used for compiled and linked object files. The extensions define the architecture of toolset compilation so that `imakef` can trace file dependencies and create the correct sequence of build commands. They are also used to deduce the transputer type and error mode for each unit.

For details of the file extensions that you must use with the `imakef` tool see the appendices of the accompanying *Toolset Reference Manual.*

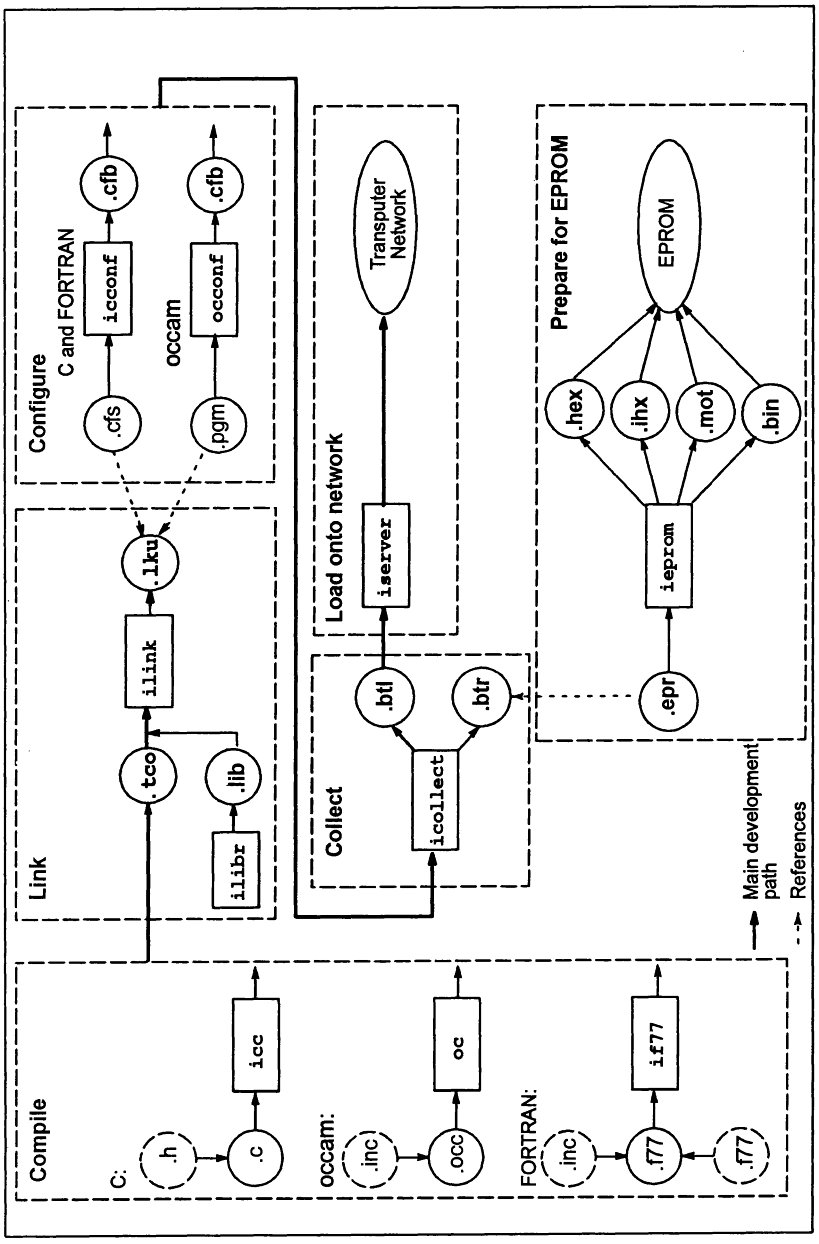


Figure 3.2 Development cycle

3.9 Error reporting

If a tool detects an error in its input, it is reported in a standard format. This contains the name of the tool, a severity level, and some explanatory text explaining why the error occurred. Errors found in files or the file system may also generate a file-name and line number. Standardization of the format is designed to improve error reporting and to support automated error handling by host system utilities.

For example:

```
Serious-ilibr-mymod.txt-bad format: not a TCOFF file
```

where: `mymod.txt` is the name of the input file causing the problem.

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

Details of the standard format can be found in the appendices of the accompanying *Toolset Reference Manual.*

3.10 Host dependencies

The toolset uses a host to develop code which is then down loaded onto a transputer or transputer network.

The toolset can be hosted on one of several different platforms, and the tools are designed to blend in as far as possible with the operating system. Source and object code is portable between all systems.

The toolset is available for the following host systems:

- IBM 386 PC (and compatibles) running MS-DOS
- Sun 4 running SunOS
- VAX running VMS.

Differences between the operation of the tools on the various platforms are minor and reflect the 'flavor' of the particular operating system.

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

Command line syntax

The major difference between host implementations is the use of the host system option prefix. For UNIX based toolsets (Sun 4) the prefix character is the dash '-';

for MS-DOS and VAX/VMS based toolsets the prefix character is the forward slash '/'.

For consistency between implementations, the case of options is not significant. However, the host syntax for filenames is used (see below), which means that on UNIX systems the case of filenames *is* significant.

Other command line syntax conventions are identical in all implementations and are described in the appendices of the accompanying *Toolset Reference Manual.*

3.10.1 Filenames

Filenames, with or without a directory path, conform to the normal host system conventions *except* that characters which can be interpreted as directory separators (on any of the supported hosts) must not be used in filenames. This prohibits the use of the following characters: colon ':', semi-colon ';', forward slash '/', backslash '\' ('¥' for Japanese systems), square brackets '[]', round brackets '()', angle brackets '<>', exclamation mark '!',or the equals sign '='.

In addition the linker cannot handle filenames which begin with a hash '#' or with two dashes '--'. These are used to identify commands and comments within linker indirect files.

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.

3.10.2 Search path

All tools which use or generate filenames use a standard mechanism for locating files on the host system. The same mechanism is used in all operating system versions of the toolset. Briefly, the search mechanism is based on a list of directories to be searched in sequence.

If a directory path is specified only this directory is searched. If the file is not found on the path an error is generated. Relative pathnames are treated as relative to the current directory, i.e. the directory from which the tool is invoked.

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

Details of how to set up a search path on your system can be found in the Delivery Manual that accompanies the release.

Full details of the mechanism used in file searching can be found in the appendices of the accompanying *Toolset Reference Manual.*

3.10.3 Environment variables

The toolsets use a number of environment variables on the host system. Use of these variables is optional but if defined they will influence the behavior of certain of the tools on your system. Further information is given in the accompanying *Toolset Reference Manual.*

Variable	Meaning
`ICONDB`	Defines the connection database to be used by `iserver`.
`ISESSION`	Defines the session manager configuration file to be used by `iserver`. Defaults to `session.cfg` if not defined.
`ISEARCH`	The search path; i.e. the list of directories that will be searched if a pathname is not specified. Pathnames must be terminated by the standard directory separator character for the system. Used by all tools that read and write files.
`ISIMBATCH`	Used by `isim` to enable/disable batch mode. Values can be `VERIFY` or `NOVERIFY`.
`ITERM`	The file that defines terminal keyboard and screen control codes. Used by `idebug`, `isim` and `iemit`.
`IBOARDSIZE`	The size (in bytes) of memory on the transputer board. Used when loading non-configured programs.
`TRANSPUTER`	Defines the capability (user link name) to be used by the server. Can be overridden by `iserver` command line option.
`IDEBUGSIZE`	The size (in bytes) of memory connected to the root transputer. Used by `idebug`.
toolname`ARG`	Default command line arguments. Applies to certain tools only. See section 3.10.4.

Table 3.2 Toolset environment variables

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

For `IBOARDSIZE` and `IDEBUGSIZE` the value can be given in decimal or hexadecimal format. Hexadecimal numbers must be preceded by '`#`' or '`$`'. Leading and trailing spaces may not be given.

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

3.10.4 Default command line arguments

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

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

Tool	Variable
`icc`	`ICCARG`
`if77`	`IF77ARG`
`ilink`	`ILINKARG`
`icconf`	`ICCONFARG`
`icollect`	`ICOLLECTARG`
`ilibr`	`ILIBRARG`
`ilist`	`ILISTARG`

Table 3.3 Environment variables for invoking tools

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

3.11 Linker startup and indirect files

Linker indirect files are text files containing lists of input files and commands to the linker.

A number of linker indirect files are supplied with each toolset. The purpose of these files is to reference various runtime libraries (or in the case of occam, compiler libraries) required to link application programs. When specifying the program modules to be linked, the appropriate linker indirect file must be included on the linker command line, as described in the reference chapter for **`ilink`** in the accompanying *Toolset Reference Manual.*

3.11.1 ANSI C Toolset

For C the linker indirect files are known as 'linker startup' files. They reference runtime library files which provide the runtime environment for the program and define which version of the C runtime initialization code is used by specifying a main entry point. This is the name of the routine which is called by the transputer bootstrap code or configuration system code, in order to start the C program executing.

Most C programs will require one of the three linker startup files listed in table 3.4. Two files are provided for use with configured programs; one with the full runtime library and one with the reduced runtime library. The reduced library does not support host I/O. It is recommended that all programs are configured.

The third file is provided for use with non-configured programs using the full runtime library.

Special linker startup files which do not specify a main entry point are described in section 3.11.4 below.

Startup file to support:		
Configured programs		Non-configured programs
Full runtime library	Reduced runtime library	Full runtime library
`cstartup.lnk`	`cstartrd.lnk`	`cnonconf.lnk`

Table 3.4 C startup files

`cstartup.lnk`

This linker startup file is used to create linked units which use the full C runtime library and are to be configured using `icconf`. It also specifies a main entry point of `C.ENTRYD`. This is the main entry point of the standard C startup code for configured systems using the full runtime library. `C.ENTRYD` is the first of a sequence of routines which are responsible for setting up the full version of the C runtime system and eventually calling the `main` function. The source of this startup code is supplied with this toolset and is described in the *ANSI C Toolset Language and Libraries Reference Manual.*

`cstartup.lnk` includes `clibs.lnk` (see 3.11.4). `cstartup.lnk` should only be used if the configurer is also used. The effect of using this linker startup file to create a linked unit which is then passed directly to `icollect`, without using the configurer first, is undefined. It should only be used when the C linked unit created is to have access to host link channels. The startup code assumes that a server exists and will attempt to communicate with it. Thus the effect of its use in an environment where there is no access to the server is undefined.

`cstartrd.lnk`

This linker indirect file is used to create linked units which use the reduced C runtime library and are to be configured using `icconf`. It also specifies a main entry point of `C.ENTRYD.RC`. This is the main entry point of the standard C startup code for configured systems using the reduced runtime library. `C.ENTRYD.RC` is the first of a sequence of routines which are responsible for setting up the reduced version of the C runtime system and eventually calling the `main` function. The source of this startup code is supplied with this toolset and is described in the *ANSI C Toolset Language and Libraries Reference Manual.*

`cstartrd.lnk` includes `clibsrd.lnk` (see 3.11.4). `cstartrd.lnk` should only be used if the configurer is also used. The effect of using this linker indirect file to create a linked unit which is then passed directly to `icollect`, without using the configurer first, is undefined. It should be used in situations where the C linked unit created has or requires no access to the server. No host link channels are defined.

`cnonconf.lnk`

This linker indirect file is used to create linked units which use the full C runtime library and are suitable for passing directly to `icollect` thereby omitting the configuration stage. **Note:** this method of program development is only applicable to single processor programs and is not recommended for any new program development as it may be unsupported in future toolsets.

`cnonconf.lnk` specifies a main entry point of `C.ENTRY`. This is a special version of the C startup code which can derive for itself information which is normally supplied by the configurer (as such it is less efficient than the equivalent version of the startup code for configured systems and so use of the configurer is recommended).

`C.ENTRY` is the first of a sequence of routines which are responsible for setting up the full version of the C runtime system and eventually calling the `main` function. `cnonconf.lnk` includes `clibs.lnk` (see 3.11.4). `cnonconf.lnk` should only be used if the configuration stage is to be omitted. The effect of using this linker indirect file to create a linked unit which is subsequently passed to the configurer is undefined. It should only be used when the C linked unit created is to have access to host link channels. The startup code assumes that a server exists and will attempt to communicate with it. Thus the effect of its use in an environment where there is no access to the server is undefined. Indeed, omission of the configuration stage is only possible if the full library is used, therefore there is no equivalent reduced version of this linker indirect file.

3.11.2 occam 2 Toolset

For occam, one of three linker indirect files should be selected according to the target transputer type(s) used, see table 3.5.

Linker indirect file	Target transputers
`occam2.lnk`	T212/T222/T225/M212
`occama.lnk`	T400/T414/T425/T426/TA/TB
`occam8.lnk`	T800/T801/T805

Table 3.5 occam linker indirect files

Each file contains a list of occam library files which may be required to be linked, but which are additional to those explicitly referenced by the program. These include compiler libraries and support for interactive debugging. Depending on the other inputs and options specified on the command line the linker will select which libraries it requires from the supplied indirect file.

3.11.3 Mixed language programs

Mixed language programs require an appropriate linker indirect file for each language used.

For occam, one of the indirect files listed in table 3.5 is always used and when the main program is written in C, one of the files listed in table 3.4 should be used.

However, if a non-C program calls in C modules, the standard C startup files are not suitable because they define a C main entry point which would conflict with the actual main entry point of the program. In this case one of the linker files described in section 3.11.4 should be used. These linker files should also be used when incorporating a C program into an occam program as if it were an occam process.

Further information about mixed language programming is given in Chapter 11.

3.11.4 Other startup files supplied with the ANSI C Toolset

Two additional linker indirect files are supplied with the ANSI C Toolset:

Linker indirect file	Comment
`clibs.lnk`	Lists the library files required for the full library.
`clibsrd.lnk`	Lists the library files required for the reduced library.

Table 3.6 C linker indirect files referencing libraries only

Unlike the files listed in table 3.4, `clibs.lnk` and `clibsrd.lnk` do not specify a main entry point. They can be used whenever the main entry point of the program is not one of the standard C entry points, for example certain mixed language programs and when producing code which will be dynamically loaded.

`clibs.lnk` should only be used when the C linked unit created is to have access to host link channels. The effect of using in an environment where there is no access to the server is undefined.

`clibsrd.lnk` should be used in situations where the C linked unit created has or requires no access to the server. No host link channels are defined.

3.12 Unsupported options

A number of tools have various command line options beginning with '`Z`'. These options are used by INMOS for development purposes and have not been designed for users. As such they are unsupported and should not be used. INMOS cannot guarantee the results obtained from such options nor their continued presence in future toolset releases.

4 Getting started

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

A more complex programming example illustrating separate compilation can be found in Chapter 5, together with a detailed description of program development for single processor systems. Chapter 6 provides examples of multitransputer programming.

4.1 Introduction

In order to create and run a transputer executable file, this sequence must be followed:

1 The source files are compiled with the occam 2 compiler. The compiler creates from each source file an object file.

2 The object files are linked together along with any libraries required, to create a file known as a linked unit. Each linked unit contains the code and data necessary to execute as a main program.

3 The linked units are then configured onto a transputer network and a bootable program is created. In the case of a single program on a single transputer, there is a short cut available here. However, it is strongly recommended that development is made by using the full facilities of the configurer. There are many advantages to this which will become apparent as the procedures are described.

4 The program is then loaded and run from the host by using the host file server. The bootable program contains everything necessary for execution on the transputer network and it will start automatically after it has been loaded.

4.2 Running the examples

In the following examples, the programs are compiled and executed on a single T425 with 1Mbyte of memory available. If you have some other transputer, then you should make the necessary changes to the command lines and configuration file as indicated. (Command line options for specifying other transputer types are listed in appendix B of the *occam 2 Toolset Reference Manual).*

The examples assume the existence of an environment variable `TRANSPUTER` which defines the name of a User Link on which to load the program, and that a *connection database* file exists to define that User Link. See the Delivery Manual which accompanies this toolset and the **iserver** documentation (chapter 13 of the *occam 2 Toolset Reference Manual*) for more details.

The examples also assume the existence of the environment variables **ISEARCH**, **ICONDB**, and **IBOARDSIZE**. See the *Delivery Manual* for details.

The tutorial assumes that you have a *boot from link* board. If you have a *boot from ROM* board or other non-standard hardware, refer to the manufacturer's documentation.

4.2.1 Sources

Source files are held in the toolset directory **examples/manuals/simple**.

4.2.2 Example command lines

In the examples below, the command lines are written in both the UNIX form with the option character '-' , and in non-UNIX form with the option character '/' (for MS-DOS and VAX/VMS systems). Choose the one that is applicable to you.

4.2.3 Using the simulator

If there is no transputer available, then you can use the simulator **isim** to run the bootable program, provided it is built for a single T425.

4.3 The example program

The example program is contained in the file **simple.occ**. **simple.occ** reads a name from the keyboard and displays a greeting on the screen. The program uses the library **hostio.lib** and the include file **hostio.inc**. The configuration description resides in the file **simple.pgm**.

The program is illustrated below.

```
#INCLUDE "hostio.inc" -- contains SP protocol
PROC simple (CHAN OF SP fs, ts)
  #USE "hostio.lib"
  [1000]BYTE buffer :
  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 section 1.5 in the *occam 2 Toolset Language and Libraries Reference Manual* for descriptions of the routines.

Before the body of the procedure a number of variables are declared. `buffer` holds the input string, `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 Compiling the program

In order to compile the program use the following command line:

```
oc simple -t425                              (UNIX)
oc simple /t425                              (MS-DOS/VMS)
```

The compiler performs the necessary syntax, alias and usage checks, inserts code to perform run-time error checking, and creates a file called `simple.tco`. Because the source file has the default extension of `.occ` you can omit the extension on the command line.

The target processor is a T425. For information about compiling for other transputer types, see section 4.4.

By default, the compiler enables interactive debugging with `idebug` and compiles the program in HALT mode (see section 5.3.1).

4.3.2 Linking the program

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

To link the program type:

```
ilink simple.tco hostio.lib -t425 -f occama.lnk  (UNIX)
ilink simple.tco hostio.lib /t425 /f occama.lnk  (MS-DOS/VMS)
```

This program uses `hostio.lib` and various target-specific compiler libraries. `hostio.lib` is directly specified on the command line; the correct compiler libraries are referenced via the '`f`' option (see below).

Note: It is always good practice to specify to the linker what the transputer target is, since it is possible to produce code that can execute on a range of transputers and the linker must then be told which the actual target will be. In this example the chosen target is T425.

The '`f`' option introduces a *linker indirect file* which is used to link in the correct compiler libraries. For more details see Chapter 9 in the *occam 2 Toolset Reference Manual.*

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

Note: In more complex programs, libraries may be dependent on other files and libraries. To ensure all necessary libraries are linked into a program, the **`imakef`** tool may be used with a suitable MAKE program (see section 4.5).

4.3.3 Configuring the program

In order to configure the program, a description is required of the network it is run on. The file **`simple.pgm`** contains such a description.

You should look at this file and edit it if it does not correspond to the hardware you actually have. For example check which link is connected to the host, the transputer type, and memory size.

The file **`simple.pgm`** contains the following:

```
NODE p :
ARC hostarc :
NETWORK
  DO
    SET p(type, memsize := "T425", 1024 * 1024)
    CONNECT p[link][0] TO HOST WITH hostarc
:

#INCLUDE "hostio.inc"
#USE "simple.lku"

CONFIG
  CHAN OF SP fs, ts :
  PLACE fs, ts ON hostarc :
  PROCESSOR p
    simple(fs, ts)
:
```

In order to configure the application for the network, the configurer is invoked as follows:

```
occonf simple.pgm
```

This produces a file called `simple.cfb` which contains all the information about where the different parts of the program are to be placed.

4.3.4 Collecting the program

The final next stage is to collect all the parts of the program and combine them into a file which can be loaded onto the transputer for execution. This is done by the collector tool `icollect`. The collector is invoked as follows:

```
icollect simple.cfb
```

The result is the executable ('bootable') file `simple.btl`.

4.3.5 Running the program on a transputer board

To load the bootable program onto a transputer board and run it use the host file server tool `iserver`:

```
iserver -se -sb simple.btl                    (UNIX)

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

The address of the transputer board is taken from the `TRANSPUTER` environment variable.

The '`sb`' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The '`se`' option directs the server to terminate if the program sets the error flag. For further information about server options see Chapter 13 in the *occam 2 Toolset Reference Manual.*

Figure 4.1 shows an example of the screen display obtained by running `simple.btl` on a UNIX based toolset, for a user called 'John'.

```
iserver -se -sb simple.btl

Please type your name :John
Hello John
```

Figure 4.1 Example output produced by running `simple.btl`

4.3.6 Running the program using `isim`

To run the example program via the simulator use one of the following commands:

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

The '`bq`' option directs the simulator to run the program in 'batch quiet' mode, that is, run the program immediately and then terminate. For more information about the simulator tool see Chapter 14 in the *occam 2 Toolset Reference Manual.*

4.3.7 A short cut to creating a bootable file

For single-transputer programs booted from transputer links attached to a host, an alternative method can be used to create the `.btl` file. This method is not applicable to standalone systems nor to systems which boot from ROM, and requires the program to be contained within a *single linked unit.* The Advanced Toolset debugger cannot be used to debug programs created in this way.

Note: Non-configured programs require a different procedural interface to configured programs because certain parameters are expressed in the configuration description. `simple.occ` would therefore require the following change:

replace:

```
PROC simple (CHAN OF SP fs, ts)
```

with:

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

Single processor programs must always use a similar parameter list. A modified version of the program can be found in the `examples` directory under the name `simple3.occ`.

To make use of the short cut, compile and link the `simple3.occ` in the same way as in the previous example. Then, omitting the configurer stage, invoke the collector directly on the linked unit, adding the `t` option to the command line (the `t` option directs the collector to build a bootable file from a single linked unit):

```
icollect simple3.lku -t                          (UNIX)
icollect simple3.lku /t                          (MS-DOS/VMS)
```

The bootable file `simple3.btl` is created. This can be loaded and run in the usual way using `iserver` or `isim`.

Note: This facility acts as a compatibility mode with previous versions of the toolset and may be discontinued in future releases.

4.4 Compiling and linking for other transputer types

If you are using a transputer other than a T425 you should specify the appropriate target transputer type for the compilation and linking operations. Appendix B in the *occam 2 Toolset Reference Manual* describes the options available. The same option must be specified to both the compiler, linker, and configurer, otherwise an error is reported. In addition, you must specify the correct linker indirect file for the selected target, in order to link in the correct compiler libraries (see Chapter 9 of the *occam 2 Toolset Reference Manual*).

For example, to compile and link the program '`simple.occ`' so that it will run on a T800, T801 or T805:

UNIX:

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

MS-DOS/VMS:

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

Modify `simple.pgm` to match the transputer type and memory size of your hardware and run `occonf` on the modified file. Then collect and load the program as before.

4.5 Using `imakef`

As an alternative method of program development the toolset Makefile generator `imakef` can be used. This tool can produce a Makefile for any type of file that can be built with the toolset tools. See Chapter 11 in the *occam 2 Toolset Reference Manual* for a description of the tool.

`imakef` serves two purposes:

- It enables the user to generate a target file automatically (e.g. a bootable file) without having to manually perform the intermediate stages of program development i.e. compiling, linking, configuring etc.
- For more complex programs, comprising several modules, it simplifies the incorporation of changes to the program by identifying dependencies and incorporating them into the Makefile.

In order for `imakef` to be able to identify file types, a different system of file extensions must be used to that used in the examples above. See section 11.3 in the *occam 2 Toolset Reference Manual* for a description of the system.

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

```
imakef simple2.btl
```

The `.btl` extension directs **imakef** to build a bootable file from a configuration description file (`simple2.pgm`). This file can be found in the **examples** directory. Within the `.pgm` file the correct format of file extension is used to reference the linked unit for **imakef**. For example:

```
#USE "simple.c5h"
```

directs **imakef** to compile the program for a T425 in HALT error mode. For other transputer types and error modes use different suffixes, see section 11.3 in the *occam 2 Toolset Reference Manual.*

To build the program run your MAKE program on **simple2.mak**:

```
make -f simple2.mak                    (UNIX)

make /f simple2.mak                    (MS-DOS/VMS)
```

This creates the bootable file `simple2.btl` which can be run in the normal way using **iserver** or **isim**.

5 Programming single transputers

This chapter provides an introduction to occam programming on single transputers. For information about programming multitransputer networks see Chapter 6.

Before reading this chapter the user should already be familiar with the concepts and syntax of the occam programming language. *A tutorial introduction to occam programming* is a good introductory text and the *occam 2 Reference Manual* contains a formal definition of the language.

5.1 Program examples

This chapter uses a relatively complex programming example (section 5.11), illustrating separate compilation. A simple programming example, to get you started, is provided in Chapter 4.

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

5.2 occam programs

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

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

The procedure and its parameters can have any legal occam names. You must always supply the procedure with the same type of formal parameters as shown above, to enable communication with the host and to enable the program to use free memory. The optional parameter `stack.buffer` may also be supplied; this allows the program to make use of the transputer's internal RAM (see sections 3.3 and 3.4.2 in the *occam 2 Toolset Reference Manual*).

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 the occam 2 *Toolset Language and Libraries Reference Manual*. 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 5.1 shows how these channels are connected.

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

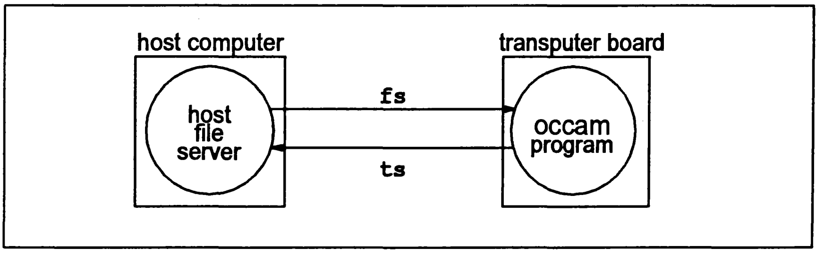


Figure 5.1 Program input/output

5.2.1 Compiling programs

The compiler produces object code in TCOFF format for input to the linker. The compiler is capable of compiling code for any one of a range of transputers (the IMS T212, M212, T222, T225, T400, T414, T425, T426, T800, T801 or T805) in one of three error modes and with interactive debugging either enabled or disabled. The compiler enables interactive debugging by default unless the compiler '`Y`' option is used.

Transputer types and classes are described in Appendix B of the occam *2 Toolset Reference Manual*.

The standard error modes are HALT system and STOP process. A special mode, UNIVERSAL, enables code to be compiled so that it may be run in either HALT or STOP mode. The target processor and error mode must be specified for each compilation, using options on the command line.

By default the compiler compiles for an IMS T414 in HALT mode, and when compiling for this transputer type and error mode you may omit the options. In all other cases the options must be supplied.

Other operating features of the compiler may be changed by options and directives. See Chapter 1 in the *occam 2 Toolset Reference Manual*.

If the compiler detects any errors, a source file name and line number is displayed along with an explanatory message and a portion of the source code surrounding the error.

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

Compilation information

It is sometimes necessary to check how much workspace (data space) will be required to run the code. This information is stored in the code file produced by the compiler, linker and librarian. To display the information use the '`I`' command line option or use the binary lister tool `ilist`. For details of `ilist` see Chapter 10 in the *occam 2 Toolset Reference Manual*.

5.2.2 Linking programs

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

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

Three linker indirect files listing the compiler libraries are supplied for different transputer types. `occam2.lnk` is provided for the T2 series, `occam8.lnk` for the T8 series and `occama.lnk` for other 32-bit transputers. The relevant file should be included on the linker command line using the '`f`' option. For further details of the compiler libraries see the *occam 2 Toolset Language and Libraries Reference Manual*.

By default, the order in which the code modules are specified on the command line determines their order within the linked unit; library modules being placed after the separately compiled modules. This default can be overruled by using the compiler directive `#PRAGMA LINKAGE` and the linkage directive `#section` (see sections 1.12.7 and 9.4.6 in the *occam 2 Toolset Reference Manual*). These directives enable the user to prioritize the order in which modules are linked together and so influence the use of on-chip RAM.

5.2.3 Displaying the contents of code files

Object code files produced by compiling or linking programs can be examined using the binary lister tool `ilist`. Information that can be displayed includes procedure definitions, exported names, external references within the code, and symbol data. For more details see Chapter 10 in the *occam 2 Toolset Reference Manual.*

5.2.4 Making bootable programs

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

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

5.2.5 Loading and running programs

Bootable programs can be loaded onto the transputer evaluation board using the host file server `iserver` (see Chapter 13 in the *occam 2 Toolset Reference Manual*). The server must be given a number of parameters when it loads a program. All server options are two characters long, with '`S`' as the first character. Server parameters are removed from the command line by the server, so you should avoid using the same options for your own program (it is best to avoid giving programs two letter options beginning with the letter '`S`').

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

Programs can also be loaded onto transputer networks, without using code on the root transputer, by first using the `iskip` tool to set up a skip process and then loading the program using `iserver`. This can be useful when loading programs onto external networks via a transputer evaluation board. It is also useful for debugging programs that normally use the root transputer to run all or part of a program. The debugger always runs on the root transputer. Provided the network has at least one processor which is not used by the program, `iskip` may be used in conjunction with `iserver` to load the program over the root transputer. Further details can be found in the *occam 2 Toolset Reference Manual.*

5.2.6 Interrupting programs

To interrupt an application program while it is still running, press the host system BREAK key to interrupt the server. See the section entitled 'Server Interrupts' in the *Delivery Manual* for the correct key to use on your system.

When the BREAK key is pressed on DOS and VAX/VMS systems 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`'.

5.3 occam error handling

For systems that require maximum security and reliability, the error behavior is of great concern. occam 2 specifies that run-time errors are to be handled in one of three ways, each suitable for different programs. The error mode to be used is supplied as a parameter to both the compiler and linker. The options are listed in Table 5.1.

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

Table 5.1 Compiler and linker options for selecting error mode

5.3.1 Error modes

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

Note: on the IMS T414, T212, T222, and M212, HALT mode does not work for processes running at high priority, as the **HaltOnError** flag is cleared when going to high priority.

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

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

All separately compiled units for a single processor must be compiled and linked with compatible error modes. HALT and STOP modes are mutually exclusive whereas UNIVERSAL mode can be mixed with either HALT or STOP.

If no mode is specified the linker defaults to HALT mode; if the program contains STOP modules then an error is generated. Similarly, if STOP is specified on the command line the presence of HALT modules generates an error.

Where a library is used the module with the appropriate error mode is selected by the compiler.

Programs may also be compiled and linked in UNIVERSAL mode. This may be useful where linked modules are used as components of the final linked program – the error mode of the program can be postponed until the final link stage which builds the whole program. Programs built entirely in UNIVERSAL mode and targetted at single processors have their error mode set by the collector tool to its default which is HALT mode.

Table 5.2 summarizes error mode compatibility.

Error mode	**Compatible with**
HALT	HALT, UNIVERSAL
STOP	STOP, UNIVERSAL
UNIVERSAL	HALT, STOP, UNIVERSAL

Table 5.2 Compatibility between error modes

Error mode UNDEFINED

The occam error mode UNDEFINED can be reproduced by specifying the `U` command line option or the `U` argument to the `#OPTION` directive.

5.3.2 Error detection

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

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

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

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

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

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

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

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

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

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

Note: The 'U option can be used to optimize runtime performance in code which is fully debugged and known to be error-free. This is equivalent to implementing UNDEFINED error mode.

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

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

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

```
PROC ASSERT (VAL BOOL test)
```

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

5.4 Interactive debugging and virtual routing

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

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

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

Disabling interactive debugging also disables virtual routing because both systems use the same set of routing and multiplexing processes.

5.5 Alias and usage checking

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

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

The behavior of programs where alias and usage checks are disabled is defined in Appendix C of the *occam 2 Toolset Language and Libraries Reference Manual.*

The '`K`' and '`U`' options will also disable the insertion of alias checks that would otherwise be performed at run-time. These options do not affect the insertion of alias checks at compile time nor the insertion of usage checks which are only performed at compile time. Alias checking can impose some code penalties, for example, extra code is inserted if array accesses are made which cannot be checked until runtime. The '`WO`' command line option will produce a warning message every time one of these checks is generated. However, alias checking can also improve the quality of code produced, since the compiler can optimize 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 according to the rules defined in the *occam 2 Reference Manual,* but with certain limitations. Normally, if it is unable to implement a check exactly, it will perform a stricter check. For example, if an array element is assigned to, and its subscript cannot be evaluated at compile time, then the compiler assumes that all elements of the array are assigned to.

If a correct program is rejected because the compiler is imposing too strict a rule, it is possible to switch off usage checking, either on the command line for the entire compilation, or by a pragma for a specific variable.

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

5.6 Using separate vector space

The compiler normally produces code which uses separate vector space. Arrays which are declared within a compilation unit are allocated into a separate 'vector space' area of memory, rather than into workspace, when they are > 8 bytes.

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

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

When a program is loaded onto a transputer in a network, memory is allocated contiguously, as shown in Figure 5.2.

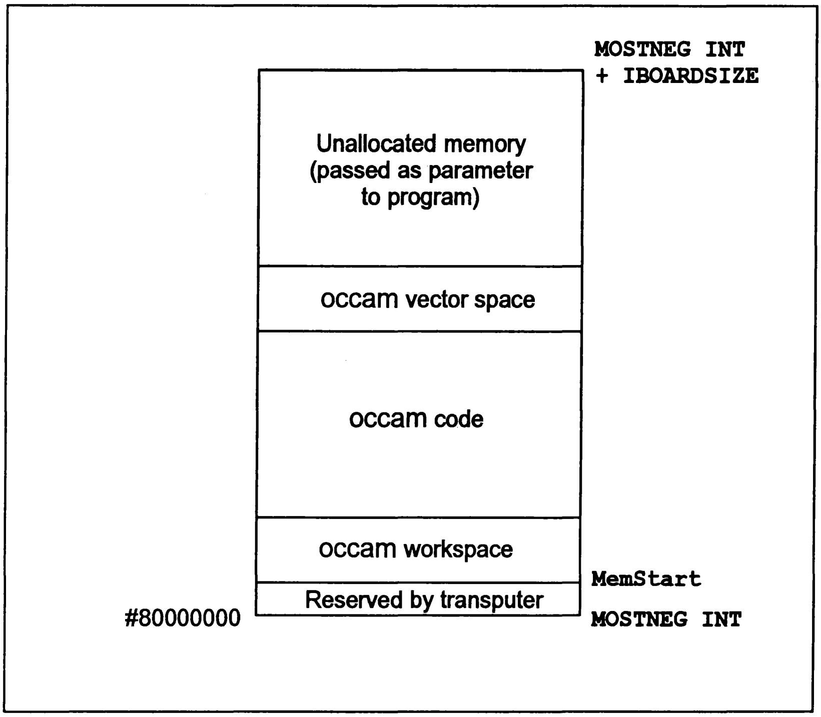


Figure 5.2 Memory allocation on a 32-bit transputer

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

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

```
PLACE name IN VECSPACE :

PLACE name IN WORKSPACE :
```

(Note: the `PLACE` statement must be inserted immediately following the declaration of the variable to which it refers).

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

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

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

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

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

5.7 Sharing source between files

The source of a program can be split over any number of files by using the `#INCLUDE` directive. This directive enables the user to specify a file which contains occam source. The contents of this file are included in the source at the same point and with the same indentation as the `#INCLUDE` directive. Include files may be nested to any depth – the compiler does not impose a limit. By convention, the `.inc` file extension is used for occam constant and protocol definitions. An example of using the `#INCLUDE` directive is given below:

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

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

5.8 Separate compilation

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

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

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

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

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

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

Separate compilation units may be nested to any depth and may contain #INCLUDE directives. They may also use libraries, as described in section 5.10. A separate compilation unit must be compiled before the source which references it can be compiled.

5.8.1 Sharing protocols and constants

occam constants and protocols may be declared and used within a compilation unit according to the rules of the language. Where a constant and/or protocol is to be used across separate compilation boundaries, it should always be placed in a separate file. The file should be referenced in any compilation unit where it is needed, by using the #INCLUDE directive before any #USE directive, which introduces procedures using the protocol in their formal parameter lists. 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.tco"

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

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

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

  SEQ
    ... body of procedure
:
```

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

5.8.2 Compiling and linking large programs

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

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

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

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

If there are a large number of input modules, they may be supplied to the linker within an indirect file, as a list of filenames. Indirect files may also contain directives to the linker. Linker directives enable the user to customize the linkage operation, e.g. define aliases, symbols, and references, modify the ordering of modules, and include other indirect files. Section 9.4 in the *occam 2 Toolset Reference Manual* describes the operation of linker directives.

5.9 Using `imakef`

When a change is made to part of a program it is necessary to recompile the program to create a new code file reflecting the change. The purpose of the separate compilation system is to split up a program so that only those parts of the program which have changed or which depend on the changed units, need to be recompiled, rather than needing to recompile the whole program. However, it would be tedious to have to remember which modules had been edited, which modules might be affected by calls and the order in which the modules were compiled and linked. For this reason a Makefile generator `imakef` is supplied with the toolset and may be used to assist with building programs consisting of several modules. This tool, when applied to a program (or part of a program), compiles a list of dependencies of compilation units and uses this list to produce a Makefile. The Makefile can be used with a suitable MAKE program to recompile only the changed parts of a program. This ensures that compilation units will always be recompiled where a change has made this necessary.

To use the Makefile generator you must tell it the name of the file you wish to build. The tool can produce a Makefile for any type of file that can be built with the toolset tools. In order for `imakef` to be able to identify file types, a different system of file extensions must be used to that used in this chapter. The filename rules for `imakef` are described in section 11.3 of the *occam 2 Toolset Reference Manual.*

5.10 Libraries

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

Libraries are referenced from occam source by the **#USE** directive. For 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.

5.10.1 Selective loading

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

The compiler is selective when a library is referenced. Only modules of a library that are of the same, or compatible, transputer type or class, error mode and method of channel input/output, are read (see Appendix B in the *occam 2 Toolset Reference Manual*, and sections 5.3 and 5.4 in this chapter).

Selective loading is based on the following rules:

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

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

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

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

Rules 1 to 3 apply to the compiler. All the rules are used by the linker. The compiler only selects on transputer type, error mode and method of channel input/output. It is not until the linking stage that unused modules are rejected. For details on mixing processor classes see Appendix B in the *occam 2 Toolset Reference Manual*, and for information on mixing error modes see section 5.3.

5.10.2 Building libraries

Libraries are built using the librarian tool **ilibr**. Libraries can be created from either separately compiled units (`.tco` or library files `.lib`) or from linked units

(`.lku` files) but not a combination of both. The librarian takes any number of input files and combines them into a single library file. Each separately compiled unit forms a single module in the library.

When forming a library the librarian will warn if there are multiply defined routines (entry points). In other words, for each combination of transputer type, error mode and method of channel input/output there may only be one routine with a particular name. For further information on building and optimizing libraries see Chapter 8 of the *occam 2 Toolset Reference Manual.*

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

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

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

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

```
ilibr -f mylib.lbb -o mylib.lib          (UNIX)

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

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

The Makefile generator `imakef` can be used to assist with the building of libraries. This is particularly useful where libraries are nested within other libraries or compilation units, because `imakef` can identify the dependencies of libraries on other modules or separately compiled units. For further information about the `imakef` tool see Chapter 11 of the *occam 2 Toolset Reference Manual.*

5.11 Example program – the pipeline sorter

This section introduces an example which serves to show how a large program might be structured, in terms of separate compilation units, libraries, and a shared protocol. occam source files, header files, and the configuration description for this program, can be found on the **examples/sorter** directory.

5.11.1 Overview of the program

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

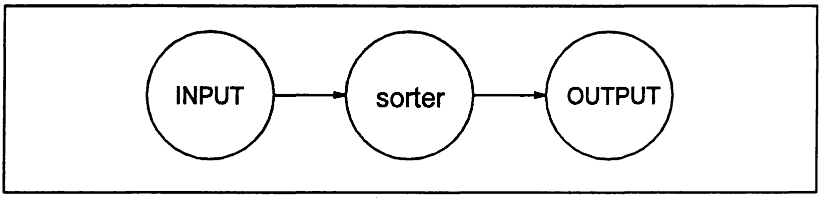


Figure 5.3 Basic structure of sorter program

Figure 5.3 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 5.4 shows the structure of the program using a pipeline.

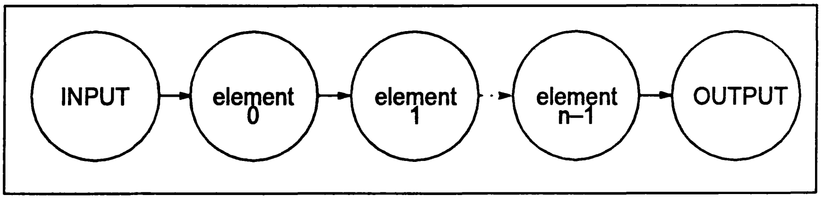


Figure 5.4 Pipeline of n elements

An obvious implementation would be to write an occam process for each process in Figure 5.4, 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 (PROCs) and as each of the processes is independent we can implement each one as a separately compiled unit. The processes share a common protocol and the best way to ensure consistency is to place the protocol in a separate file and use the

#INCLUDE mechanism to access it. These processes can then be called in parallel by an enclosing program which can access the code of each process by the #USE mechanism.

There is a problem with this implementation because two processes require access to the host file server. The host file server is accessed via a pair of occam channels and occam does not allow the sharing of channels between processes. There are a number of ways around this problem. One solution is to use a multiplexor process for the server channels, as described in section 8.5. Another solution is to merge the two processes into a single process. This solution is used because the program accesses the server in a sequential manner (read a line then display sorted line, read a line etc.). Figure 5.5 gives the final process diagram for the program.

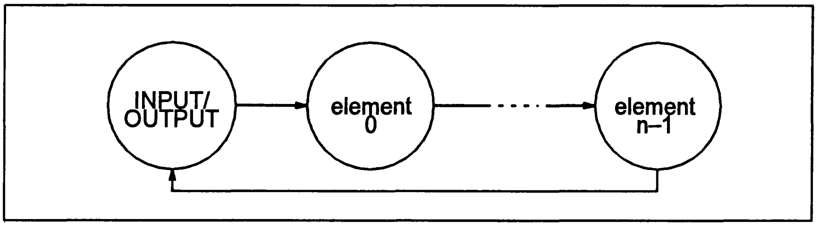


Figure 5.5 Program with combined input/output process

The implementation can be split functionally 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 5.6 shows the way these files are connected together to form a program.

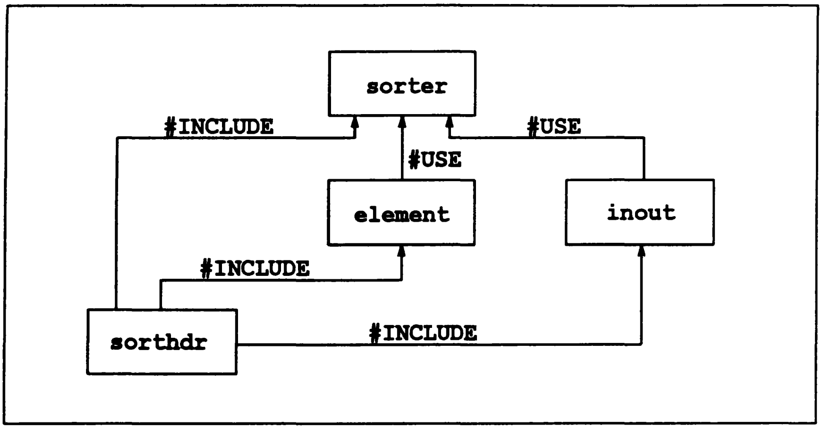


Figure 5.6 File structure of program

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.

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. These are automatically referenced using the `ISEARCH` environment variable.

5.11.2 The channel protocol

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

```
PROTOCOL LETTERS
  CASE
    letter; BYTE
    end.of.letters
    terminate
:
VAL number.elements IS 100:    -- upper bound
```

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.

5.11.3 The sorting element

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

```
#INCLUDE "sorthdr.inc"
PROC sort.element (CHAN OF LETTERS input, output)
  BYTE highest:
  BOOL going:
  SEQ
    going := TRUE
    WHILE going
      input ? CASE
        terminate
          going := FALSE
        letter; highest
          BYTE next:
          BOOL inline:
          SEQ
            inline := TRUE
            WHILE inline
              input ? CASE
                letter; next
                  IF
                    next > highest
                      SEQ
                        output ! letter; highest
                        highest := next
                    TRUE
                      output ! letter; next
                end.of.letters
                  SEQ
                    inline := FALSE
                    output ! letter; highest
            output ! end.of.letters
    output ! terminate
:
```

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

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

5.11.4 The input/output process

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

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

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

```
#INCLUDE "sorthdr.inc"
#INCLUDE "hostio.inc"
PROC inout (CHAN OF SP fs, ts,
            CHAN OF LETTERS from.pipe, to.pipe)
  #USE "hostio.lib"
  [number.elements - 1]BYTE line, sorted.line:
  INT line.length, sorted.length:
  BYTE result:
  BOOL going:
  SEQ
    so.write.string.nl (fs, ts,
    "Enter lines of text to be sorted *
    *- empty line terminates")
    going := TRUE
    WHILE going
      SEQ
        so.read.echo.line(fs, ts, line.length,
                          line, result)
        IF
          result <> spr.ok
            STOP -- stop if an error occurs
          TRUE
            so.write.nl (fs, ts)
        PAR
          SEQ
            IF
              (line.length = 0)
                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
  :
```

5.11.5 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"
  #INCLUDE "sorthdr.inc" --
  #USE "inout"
  #USE "element"

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

5.11.6 Compiling the program

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

The program's components must be compiled in a bottom up fashion, that is, `element.occ` and `inout.occ` first (in either sequence), followed by the main program `sorter.occ`. First, compile the sorting element `element.occ` using one of the following commands:

```
oc element -t425                              (UNIX)
oc element /t425                              (MS-DOS/VMS)
```

The file extension can be omitted on the command line because the source file has the conventional extension `.occ`. The compiler produces a file called `element.tco`, compiled for a T425 in HALT mode.

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

```
oc inout -t425                                (UNIX)
oc inout /t425                                (MS-DOS/VMS)
```

The compiler produces a file called `inout.tco`, compiled for a T425 in the default HALT error mode.

Finally compile the main body using the command line:

```
oc sorter -t425                               (UNIX)
oc sorter /t425                               (MS-DOS/VMS)
```

The compiler produces a file called `sorter.tco`, compiled for a T425 in HALT mode.

5.11.7 Linking the program

Having compiled all the components of the program you can now link them together to form a whole program. Any libraries used by the program must also be specified to the linker. The library `hostio.lib` is the server library used by this program. Remember the include file, `occama.lnk`, which identifies the other libraries, such as compiler libraries, required in the linking process (see section 3.11.2).

To link the files use one of the following commands:

```
ilink sorter.tco inout.tco element.tco hostio.lib -f occama.lnk -t425
                                                  (UNIX)
ilink sorter.tco inout.tco element.tco hostio.lib /f occama.lnk /t425
                                                  (MS-DOS/VMS)
```

The linker will create the file `sorter.lku` linked for a T425 in HALT mode.

If a main entry point is not specified, the linker uses the first valid entry point that it encounters in the input. Therefore, in the above example, it is important to list the file '`sorter.tco`' first. A main entry point may be specified within an indirect file using the linker directive `#mainentry` or on the command line using the linker '`ME`' option.

5.11.8 Configuring and collecting the program

Before you can run the program you must configure and collect the program. This will generate a bootable file which can be loaded and run using `iserver` or `isim`. Use the following sequence of commands:

```
occonf sorter.pgm

icollect sorter.cfb
```

`occonf` generates the file `sorter.cfb` which is then processed by the collector tool This creates the bootable file `sorter.btl`.

`sorter.pgm` configures the program for a single IMS T425 with 1Mbyte of memory; this should be checked against your own hardware and modified if necessary:

```
NODE p :
ARC hostarc :
NETWORK
  DO
    SET p(type, memsize := "T425", 1024 * 1024)
    CONNECT p[link][0] TO HOST WITH hostarc
:

#INCLUDE "hostio.inc"
#USE "sorter.lku"

CONFIG
  CHAN OF SP fs, ts :
  PLACE fs, ts ON hostarc :
  PROCESSOR p
    [1]INT dummy.memory :
    sorter(fs, ts, dummy.memory)
:
```

5.11.9 Running the program

The bootable file can be run using `iserver` or `isim`. To load the program onto a transputer board use one of the following commands:

```
iserver -se -sb sorter.btl            (UNIX)

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

The '`sb`' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The '`se`' option directs the server to terminate if the program sets the error flag. For more details about running the iserver see Chapter 13 in the *occam 2 Toolset Reference Manual.*

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

Figure 5.7 shows an example of the screen display obtained by running `sorter.btl` on a UNIX based toolset. The user inputs the string 'Sorter program' and terminates the program by pressing RETURN.

```
iserver -se -sb sorter.btl
Enter lines of text to be sorted - empty line terminates
Sorter program
 Saegmooprrrrt
```

Figure 5.7 Example output produced by running `sorter.btl`

To run the program using `isim` use one of the following commands:

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

The '`bq`' option specifies batch quiet mode which causes the simulator to run the program and then terminate. For a description of the simulator tool see Chapter 14 in the *occam 2 Toolset Reference Manual.*

5.11.10 Alternative method of creating a bootable file

Because the program is to be loaded on a single transputer, a shortcut may be used (see section 4.3.7). The bootable file can be created directly from the linked unit using the collector '`t`' option, omitting the configurer stage:

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

The '`t`' option informs the collector that the input file is a linked unit rather than the output of the configurer tool. If this method is used, the collector creates as a by-product a `.cfb` file, redundant in this example.

As in the configured case the collector creates the bootable file `sorter.btl` which can be loaded and run using `iserver` or `isim`.

5.11.11 Automated program building

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

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

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

`imakef` requires the user to adopt a particular convention of file extensions. The user then only has to specify the target file s/he requires i.e. a bootable file and `imakef`, using its knowledge of file names rules, creates a suitable Makefile. This file has full instructions on how to build the program. By running the MAKE program for the file the entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

For more details about the `imakef` tool and an example of how to create a makefile for the pipeline sorter program used in this chapter, see Chapter 11 in the *occam 2 Toolset Reference Manual.*

6 Configuring transputer networks

This chapter describes how to build programs that run on networks of transputers. It describes how to configure an occam program for a network of transputers using the configuration language and the occam configurer tool `occonf`, illustrated with an example program for four transputers. The chapter also includes examples illustrating various aspects of configuration.

6.1 Introduction

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

Programs consisting of occam processes can be run on single or multiple transputers, in any combination. Performance requirements can be met by adapting the application to run on differing numbers of transputers, and by using differing network topologies.

The mapping of processes to processors on a transputer network is known as configuration. Transputer programs can be configured to run on any physical network of transputers. They can be loaded from an external host down a transputer link, or loaded from ROM.

Configuration is achieved by writing a configuration description in the occam configuration language. A configuration description is created by the user as a text file which is processed by the configurer tool to generate a configuration data file. This data file is in turn processed by the collector tool `icollect` to generate a transputer loadable file.

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

6.1.1 Mixing languages

By using the facilities for calling other languages from occam, programs compiled from mixed language sources may also be configured using the occam configurer. (These facilities enable the foreign language code to be incorporated into the occam program as equivalent occam processes. An example of this is provided in the user `examples` directory supplied with the toolset. A description of this method of mixed language programming is given in Chapter 11). Similarly it is possible to configure occam modules which are called by C programs using the configurer provided with the ANSI C toolset. Details of how to do this are given in the *ANSI C Toolset User Guide.*

6.2 Configuration model

The configuration model consists of the following parts:

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

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

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

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

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

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

6.2.1 Configuration language

A configuration description consists of a sequence of declarations and statements. The language used is an extension to occam and follows the usual occam scope rules – in fact, the configurer uses the occam compiler to evaluate these statements. Appendix A defines the syntax of the occam configuration language.

Configuration declarations introduce physical processors, arcs and edges of the network, network connections and processor attributes, logical processors to be mapped onto physical processors, the software description, and the mapping between logical and physical processors. These are listed in Table 6.1.

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

Table 6.1 Configuration description declarations

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

Configuration declarations are usually followed by statements which perform various actions relating to the declaration. Actions are defined by `SET`, `CONNECT` and `MAP` statements. The `DO` construct enables these statements to be grouped or replicated. `PROCESSOR` statements introduce processes which may be mapped onto named processors. `IF` may be used as in occam. Configuration language statements are listed in Table 6.2.

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

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

Table 6.2 Configuration description statements

Importing code and source files

Compiled code from other files may be referenced by means of the #USE directive, either at the top level, or within the CONFIG construct.

#INCLUDE directives can be used to include other source files. It is suggested that the distinct sections are kept in different files, accessed by #INCLUDE directives from a 'master' file.

The include file `occonf.inc`, supplied with the toolset, defines some useful configuration values. It can be found on the toolset `libs` directory.

6.2.2 Overall structure of a configuration description

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

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

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

The mapping section can be used to ease the task of changing a particular program to execute on a different hardware network. The mapping section enables this to be performed without modifying the software description in any way, by flexibly mapping the *logical* processors onto the *physical* processors, see figure 6.1.

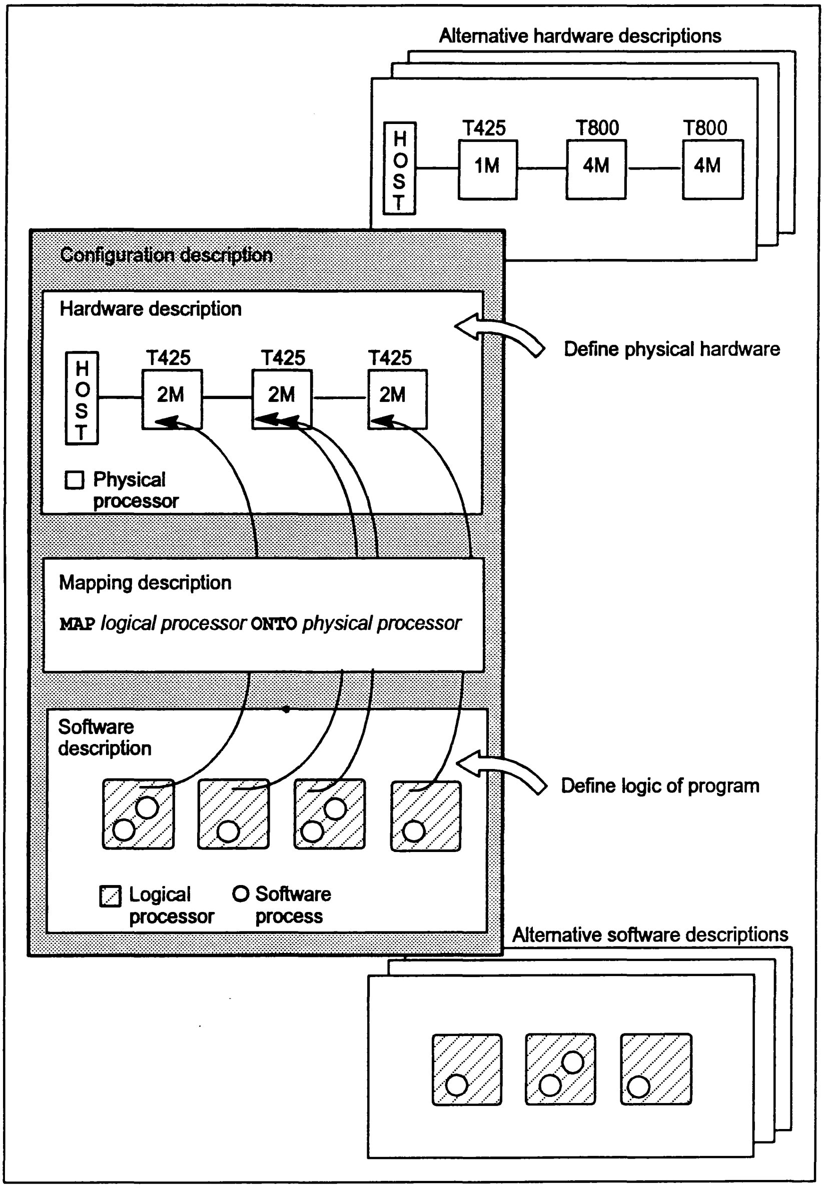


Figure 6.1 Configuration using logical processors

The following example illustrates the basic style of the language:

```
-- hardware description, omitting host connection
#INCLUDE "occonf.inc" -- contains useful constants
                      -- for memory sizes

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

-- software description
#INCLUDE "prots.inc" -- declare protocol
#USE "root.lku"      -- must be linked units
#USE "worker.lku"
CONFIG
  CHAN OF protocol root.to.worker, worker.to.root :
  PLACED PAR
    PROCESSOR root.l
      root.process(worker.to.root, root.to.worker)
    PROCESSOR worker.l
      worker.process(root.to.worker, worker.to.root)
:
```

This example is illustrated in Figure 6.2.

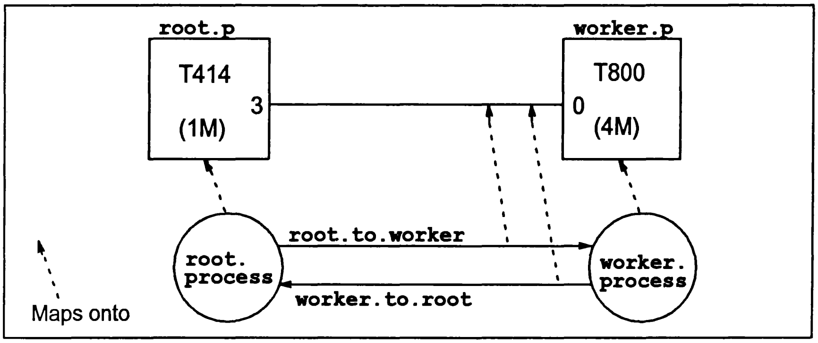


Figure 6.2 Mapping of software onto hardware

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

As an optimization, for simple programs, or for programs which will never need to be re-mapped, the software description may reference the *physical* processors directly, avoiding the need to introduce *logical* processor names.

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

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

```
-- hardware description, omitting host connection
#INCLUDE "occonf.inc" -- contains useful constants
                      -- for memory sizes

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

-- software description
#INCLUDE "prots.inc" -- declare protocol
#USE "root.lku"      -- must be linked units
#USE "worker.lku"
CONFIG
  CHAN OF protocol root.to.worker, worker.to.root :
  PLACED PAR
    PROCESSOR root
      root.process(worker.to.root, root.to.worker)
    PROCESSOR worker
      worker.process(root.to.worker, worker.to.root)
:
```

The configurer automatically maps the defined software processes to the available physical processor names.

6.3 Hardware description

6.3.1 Declaring processors

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

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

6.3.2 NODE attributes

A NODE representing a physical processor has a set of attributes, analogous to fields of a record. An attribute is referenced by subscripting the name of the node with the name of the attribute.

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

Mandatory attributes that must be set in the hardware description are as follows:

`type`	Defines processor type. Processor types supported are: T400 T414 T425 T426 T800 T801 T805 T212 T222 T225 M212
`link`	Defines processor and network node interconnections. Only defined if `type` has already been defined.
`memsize`	Defines processor memory size in bytes.

Optional attributes which can be set in the hardware description are:

`root`	Defines the root processor if there is no host connection. Takes the values TRUE or FALSE.
`romsize`	Size of ROM attached to the processor, expressed as an integer number of bytes. Mandatory if `root` is TRUE.

Additional processor attributes can be set in the MAPPING section. These support a variety of features including:

- relative ordering of code and vector space
- reservation of memory
- placement of code, workspace, and vector space
- fine-tuning of software virtual routing
- disablement of the INMOS *INQUEST* tools for specific processes.

Attributes that can be set in the `MAPPING` section are listed below.

`order.code`	Defines the priority of the program code in memory.
`order.vs`	Defines the priority of the program's vectorspace in memory.
`order.ws`	Defines the priority of the program's workspace in memory.
`reserved`	Defines a block of memory to be reserved for code placement.
`location.code`	Defines an absolute address at which program *code* should be placed.
`location.ws`	Defines an absolute address at which the *workspace* (stack) should be placed.
`location.vs`	Defines an absolute address at which the *vectorspace* should be placed (if it exists).
`routecost`	Weights or de-weights specific processors in the network for virtual routing.
`tolerance`	Defines the level of usage of a particular processor for load-sharing routing paths.
`linkquota`	Defines the maximum number of links on the processor to be used by virtual routing.
`nodebug`	For use with the *INQUEST* debugger. Informs the debugger that the process is not to be debugged. Takes the values `TRUE` or `FALSE`; the default is `FALSE`.
`noprofile`	For use with the *INQUEST* profiling tools. Informs the tools that process is not to be profiled. Takes the values `TRUE` or `FALSE`; the default is `FALSE`.

Use of these attributes is fully described in sections 6.5.5 to 6.5.9.

6.3.3 NETWORK description

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

To describe a single processor, the `SET` statement provides values for the processor's attributes in the style of a multiple assignment.

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

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

Supported types are:

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

The `memsize` attribute must be set to the amount of usable memory (on-chip + external memory) available to that processor. It is expressed as a contiguous amount starting at the most negative address, in `BYTE`s. (`K` and `M`, defined in `occonf.inc`, can be used to specify Kbytes and Mbytes).

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

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

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

Since the `DO` construct does not imply any particular ordering, there is no absolute constraint on the order in which attributes may be defined. However, it is considered good occam style by many to declare processor types and attributes before other statements such as `CONNECT` statements.

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

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

Processors must be connected together by means of `CONNECT..TO..` statements quoting a pair of edges:

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

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

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

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

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

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

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

6.3.4 Declaring EDGEs

Declared EDGEs define the ends of external connections of a NETWORK. For instance, a connection to another machine whose internal structure is irrelevant.

They are declared as though they were occam data types, and as usual we can declare arrays of them:

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


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

6.3.5 Declaring ARCs

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

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

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

6.3.6 Abbreviations

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

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

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

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

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

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

6.3.7 Host connection

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

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

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

6.3.8 Examples of network descriptions

1) *Single processor configuration connected to host:*

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

This configuration is illustrated in Figure 6.3.

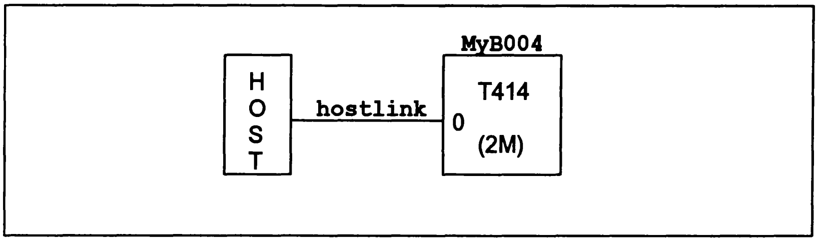


Figure 6.3 Example of host connection

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

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

This network is illustrated in Figure 6.4.

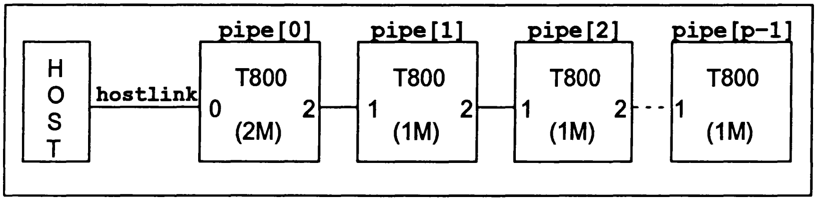


Figure 6.4 Simple pipeline with different processor memory sizes

3) *Square array with host interface processor:*

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

    DO i = 0 for p
      DO j = 0 for p
        DO
          SET Square[i][j] (type, memsize := "T800", 1*M)
          IF
            (i = 0) AND (j = 0)
              CONNECT HostSquare  [link][Down] TO
                      Square[0][0][link][Up]
            i = 0
              CONNECT Square[p - 1][j - 1][link][Down] TO
                      Square[0    ][j    ][link][Up]
            TRUE
              CONNECT Square[i - 1][j][link][Down] TO
                      Square[i    ][j][link][Up]

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

6.4 Software description

The software description is introduced by a CONFIG statement and may optionally be given a name.

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

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

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

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

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

The process 'inside' the PROCESSOR statement may consist of occam text. However, it is recommended that the code should be restricted to simple procedure calls i.e. to separately compiled procedures, referenced as linked compilation units using the #USE directive. Code which generates library calls is not allowed.

Note: when **imakef** is used to build the program, any linked units referenced by the software description must be given extensions of the type **.cxx**. This is because **imakef** uses a different convention for file extensions to the normal TCOFF file extensions, see chapter 11 in the *occam 2 Toolset Reference Manual.*

6.4.1 Libraries of linked units

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

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

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

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

6.4.2 Example

The following example of a software description, is for the pipeline sorter program introduced in section 5.11. The example is developed to show the complete configuration description for the program, in section 6.6. Figure 6.5 illustrates the mapping of the software processes onto a network of logical processors, which in this example is achieved without an actual mapping section. This method of mapping is explained in section 6.5.4.

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

This example names a single processes `inout.p` and an array of processes `pipe.element.p`. The code may be mapped onto any hardware configuration onto which matches the defined logical network and which includes an ARC declaration for the host connection `hostlink`.

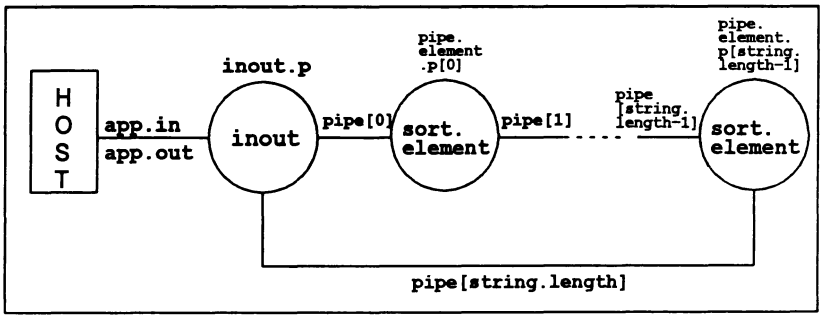


Figure 6.5 Pipeline sorter — mapping processes onto processors

6.5 Mapping descriptions

A `MAPPING` structure is used if the user has declared logical processors. The `MAPPING` maps logical processors used in the software description onto physical processors used in the hardware description. It is possible to map any number of logical processors onto any physical processor. The mapping description may also place software channels on processor links.

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

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

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

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

6.5.1 Mapping processes

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

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

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

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

This is exactly equivalent to:

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

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

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

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

The file occonf.inc includes two named constants HIGH and LOW which can be used for this.

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

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

6.5.2 Channels

Channels are unidirectional, point-to-point connections which may be implemented in one of four ways:

- *Soft* channel – a channel which communicates between processes running on the same processor.
- Channel *edge* – a channel which provides communication between the network and the outside world.
- *Direct* channel – one of up to two channels (one in each direction) placed on a single link between adjacent processors.
- *Virtual* channel – a channel placed on on a *virtual* link.

No further action is required at configuration time to define or place the *soft* channels within an application; they are fully defined by the software itself.

Channel edges *must* be placed on a *hardware arc*. This can be done with a PLACE or MAP statement:

```
PLACE fs ON hostarc:
```

or:

```
MAP fs ONTO hostarc
```

All other channels on a network may be implemented as either *direct* channels or *virtual* channels. By default the configurer *automatically* places software channels on links using the placement of processes on processors and channel edges on hardware edges as a guide.

The configurer can implement many channels over a single hardware link as well as channels between non-adjacent processors; channels implemented in this way are known as *virtual channels*. They are implemented by software virtual routing processes added automatically, as required, by the configurer.

Direct channels occur when only one or two channels (one in each direction) are placed on a link between adjacent processors. Direct channels may be automatically allocated by the configurer or the user may specifically place up to two channels on a named arc. For example, a channel *edge* is an example of a mandatory direct channel. **Note:** when interactive debugging with `idebug` and virtual routing are both enabled (the default), any direct non-edge channel placements will be ignored.

Virtual channels enable an application program to run on most network topologies irrespective of the number of physical links connecting processors. The configurer can form virtual channels that span up to 24 hops across the target network. (A 'hop' is when a processor is required for routing a channel, zero hops implies that no processors were required to route the channel). Should the configurer fail to implement a long distance connection in a very large network, it will generate an error message. Chapter 10 provides further information about routing channels.

Virtual channels are unidirectional and synchronized and are implemented by means of *virtual links*. A virtual link can be thought of as a bi-directional virtual connection between two processors, providing a communication path sufficient for two channels (one in each direction) and the appropriate synchronization signals.

Explicit placement of channels on arcs using *direct* channels is only required when connecting channels to hardware edges or where links are used for special purposes. For example, connection to a device, or where an application uses input and output channels separately, as in software implementations of high-speed links. In certain performance critical applications it may also be important to avoid the overhead incurred when using virtual channels.

A link may carry one explicitly placed channel as well as many virtual channels (in the opposite direction). In addition a pair of virtual channels (one in each direction) may be routed by the configurer via different physical links.

In general channels should not be explicitly placed on arcs, unless they are edge connections. This enables the configurer to implement channels where applicable using routing and multiplexing software.

Attention: If it is essential that the configuration does not use any virtual routing, e.g. for performance reasons, the `occonf` 'NV' command line option should be used. This disables the configurer from using the virtual routing processes. (The configurer will fail if configuration is not possible, in which case the configuration should be modified to ensure that all channels can be placed). The bootable file generated will be smaller when the virtual routing processes are not included.

6.5.3 Mapping channels

Channels between processors need not be placed by the user. The configurer will determine that a connection exists, and will allocate all the channels to links if they are available. The example in section 6.4.2 demonstrates this method and this is the simplest way of implementing virtual channels. If virtual channels are to be used it is essential that some channels are left unplaced.

However, if a user wants to override the default allocation, channels may be explicitly mapped onto named ARCs. Also, channels connecting processors to external EDGEs must be mapped onto an ARC which connects to that EDGE.

Channels are mapped onto ARCs in exactly the same way as logical processors are mapped onto physical processors. Two channels may be mapped onto the same ARC. Obviously the ARC must connect EDGEs of the processors onto which are mapped the processes which use the channel.

The channel behavior of the D7205/D5205/D6205 occam 2 toolsets may be recreated by specifying the configurer 'NV' option, which disables all virtual routing.

Channels may be assigned to arcs within the MAPPING section. For example:

```
EDGE peripheral :
ARC  peripheral.arc :
NODE root.proc :              -- physical processor
NETWORK n
  DO
    -- insert code to set attributes, then:
    CONNECT root.proc[link][0] TO peripheral WITH peripheral.arc
:
CHAN OF protocol to.periph, from.periph :
NODE process :                -- logical processor
CONFIG
  PLACED PAR
    PROCESSOR process
      -- reads from channel from.periph, writes to
      -- channel to.periph
:

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

From the above example it can be seen that more than one channel can be mapped to a single arc. This makes it easy to place two opposing channels onto a transputer link using a single line of code.

6.5.4 Mapping without a MAPPING section

Channels can also be assigned to arcs outside the `MAPPING` section, using the `PLACE` statement. This is known as *channel allocation*. Any channel in scope at the point where a process is labelled is available for explicit placement on an arc declared in the hardware network.

Placements must immediately follow the channel declaration. For example:

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

As with channel mapping, two opposing channels can be assigned to the same link in a single statement.

6.5.5 Moving code and data areas

Three processor attributes may be used to provide greater control of the layout of code and data areas in memory. Since these attributes are essentially properties of the user's program, not of the hardware description, the settings must be made as part of the `MAPPING` section. However, the processor which is referenced must be a *physical* processor.

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

These priorities can be overridden by setting three processor attributes: `order.code`; `order.ws`; and `order.vs`; which correspond to the program code, the program's workspace, and the program's vectorspace respectively. They are all set to 0 by default.

These attributes can be set to `INT` values, where lower integers indicate a higher priority. Hence setting `order.code` to –1 means that the program's code will be placed at a lower address than the workspace or vectorspace. The default ordering if priorities are equal is: workspace; code; vectorspace (workspace is placed lowest in memory).

Thus we may have a mapping section like:

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

This would place the program code before the workspace i.e. closer to on-chip RAM. In this mapping vectorspace has no priority defined and is therefore placed by default after the workspace.

All three attributes must be *enabled* on the configurer command line by the code re-ordering option '`RE`'. If this option is not specified on the command line the attributes will be ignored.

Note: Changing the default ordering means that the INMOS debugger cannot be used. It is for this reason that the attributes must be explicitly enabled.

6.5.6 Reserving memory

A block of memory may be reserved using the processor attribute `reserved`. The block is specified as a number of bytes starting at the bottom of memory. Since this attribute is essentially a property of the user's program, not of the hardware description, the setting must be made as part of the `MAPPING` section. However, the processor which is referenced must be a *physical* processor.

By default, the configurer uses memory in a contiguous block from the bottom of the transputer's available memory (near **MemStart**) to the top of the memory specified by the `memsize` attribute.

This can be overridden by means of the `reserved` attribute. This attribute specifies the number of bytes of memory which should be reserved so that the configurer does not use it. By default, this value is approximately the number of bytes below **MemStart**. It must be set to a positive value.

For example:

```
MAPPING reserve.low.memory
  DO
    MAP logical ONTO physical
    SET physical (reserved := #1000)
:
```

This would ensure that the bottom 4096 bytes of memory are reserved and will not automatically be used by the configurer.

Use of the `reserved` attribute is described in more detail in section 10.1.

6.5.7 Absolute address code placement

The `location.` processor attributes allow various parts of a program to be placed at absolute addresses in the transputer's address space. Since these attributes are essentially properties of the user's program, not of the hardware description, the settings must be made as part of the `MAPPING` section. However, the processor which is referenced must be a *physical* processor.

The address referenced by `location` must not have been used by the configurer's normal scheme; i.e. it must either lie in an area reserved by the `reserved` attribute, or must be above `memsize` bytes from the bottom of memory.

There are three location attributes:

`location.code`	specifies the absolute address at which program *code* for this processor should be placed.
`location.ws`	specifies the absolute address at which the *workspace* (stack) for this processor should be placed.
`location.vs`	specifies the absolute address at which the *vectorspace* for this processor should be placed (if it exists).

For example:

```
MAPPING use.absolute.addresses
  DO
    MAP logical ONTO physical
    SET physical (reserved    := #1000)
    SET physical (location.ws := #80000100)
:
```

This would ensure that the bottom 4096 bytes of memory are reserved and will not automatically be used by the configurer, except that the workspace is placed at address `#800000100`. (This is just above **MemStart**, in the transputer's on-chip RAM).

```
MAPPING use.absolute.addresses
  DO
    MAP logical ONTO physical
    SET physical (location.code := #80001000)
    SET physical (location.vs   := #80002000)
:
```

This would place the code at address `#800001000`, and the vectorspace at `#80002000`.

`location` attributes must be enabled on the configurer command line using the'`RE`' option. If this option is not given these attributes will be ignored.

Use of the `location` attributes is described in more detail in section 10.1.

6.5.8 Control of routing and virtual channel placement

Three processor attributes can be used to control the way that virtual routing is performed. They are `routecost`, `tolerance`, and `linkquota`. Since these attributes are essentially properties of the user's program, not of the hardware description, the settings must be made as part of the `MAPPING` section. However, the processor which is referenced must be a *physical* processor.

The exact behavior of these attributes is and their use in defining routing strategy for a network is described in section 10.2.4.

Values of all three attributes are defined as numerical values. For example:

```
MAPPING routing.example
  DO
    MAP logical ONTO physical
    SET physical (routecost := 1000)
    SET physical (tolerance := 1000)
    SET physical (linkquota := 2)
:
```

`routecost`:

This attribute is used to weight and de-weight specific processors in the network for virtual routing. It defines an associated cost of routing virtual channels through a particular processor. `routecost` can be used to specifically exclude certain processors from the virtual routing network.

`routecost` may be set to an `INT` value within the range 1 to 1000000 inclusive. If a value greater than 1000000 is specified, then no through-routing will be permitted on that processor. If `routecost` is not specified for a particular processor, then a default cost value of 1000 is assumed.

`MIN.COST`, `MAX.COST`, `INFINITE.COST` and `DEFAULT.COST` are defined in the include file `occonf.inc`, see below.

`tolerance`:

This attribute is used to indicate how much a particular processor can be used to provide load-sharing routing paths for other processors.

`tolerance` may be set to an `INT` value within the range 0 to 1000000 inclusive. If `tolerance` is not specified for a particular processor, then the default value of 1 will be assumed. This allows the processor to implement alternative routes for through-routed channels with exactly the same cost as the "best" route found between any two other processors.

If the value 0 is specified, then the processor will only be used for through-routing if it lies on the "best" route found to implement virtual channels.

If `tolerance` is set to the maximum value 1000000 on all processors in the target network almost every possible route will be used to share the cost of carrying data between any pair of non-adjacent processors.

`ZERO.TOLERANCE`, `MAX.TOLERANCE`, and `DEFAULT.TOLERANCE` are defined in the include file `occonf.inc`, see below.

`linkquota`:

This attribute is used to indicate the maximum number of links on the processor that should be used by the virtual channel routing system.

`linkquota` may be set to an `INT` value within the range 0 to 4 inclusive. A warning is generated if the suggested `linkquota` for a processor is exceeded. This will only happen if it is necessary for through-routing other processors.

`occonf.inc`

This file contains a number of constants associated with the routing and placement attributes. The file must be included within the configuration description if any of the configuration constants are used e.g.

```
#INCLUDE occonf.inc
```

6.5.9 Control of debugging by the *INQUEST* tools

Two attributes are included for use only with the INMOS *INQUEST* product, namely, `nodebug` and `noprofile`. These are boolean parameters which control the *INQUEST* debugging and profiling tools respectively. They can be set to `TRUE` or `FALSE`. If set equal to `TRUE` for a process, that process will be ignored. The default for both attributes is `FALSE`.

`nodebug` and `noprofile` have no effect on the functioning of the toolset debugger `idebug`, or on any other tool supplied with the current toolset.

6.5.10 Mapping examples

1) *pipeline sorter on a single processor*

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

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

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

6.6 Example: A pipeline sorter on four transputers

This section describes how the pipeline sorter program first described in section 5.11 may be distributed over four T425 transputers. Each processor has many processes allocated to it.

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

The occam source and configuration description developed in this example is supplied with the toolset in the `examples/manuals/sorter` directory; you can either work in this directory or copy the relevant files to a working directory:

`sorthdr.inc`	the common protocol definition.
`element.occ`	the sorting element.
`inout.occ`	the interface to the host file server.
`sortconf.pgm`	the configuration description for the network.

`sorthdr.inc`, `element.occ`, and `inout.occ` are the same as those used in the single transputer example described in section 5.11.

`sortconf.pgm` describes the hardware and software networks and maps the software to the hardware. The software description is imported in the include file `sortsoft.inc`.

In the configuration description it is assumed that there is a transputer network of four T425 transputers connected in the pipeline configuration shown in Figure 6.6. If this configuration does not match your hardware the description can easily be modified by changing the number and type of transputers. The example assumes the link connections shown in Figure 6.6.

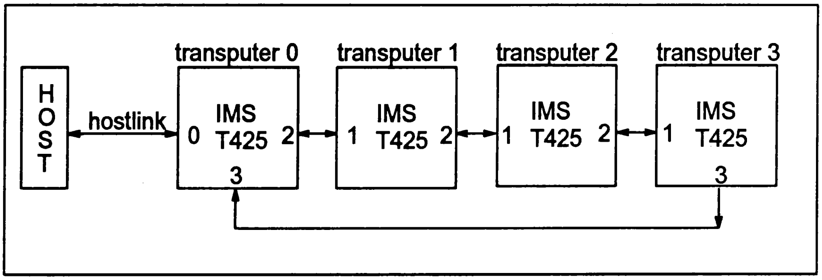


Figure 6.6 Pipeline of four transputers

The mapping places an equal number of element processes on all processors in the pipeline after the first one, which gets any remaining element processes.

```
--To run this program connect any number of identical
--transputers in a pipeline: connect link 1 of processor 0
--to link 2 of processor 1 and sq on; complete the pipeline
--by connecting first and last via their link 3s;
--finally connect processor 0 to the host.

--max no of chars on a line
VAL string.length IS 80:
--include useful definitions e.g. K=Kilo, M=Mega
#INCLUDE "occonf.inc"
--change the following to suit your network
VAL number.of.transputers IS 4:
--declare processors as an array
[number.of.transputers]NODE pipeline.t:
ARC Hostlink:

--hardware description
NETWORK
  DO
    DO i = 0 FOR number.of.transputers
      --change the following to suit your transputer type
      SET pipeline.t[i] (type, memsize := "T425", 1*M)

    DO i = 0 FOR number.of.transputers - 1
      CONNECT pipeline.t[i][link][2] TO pipeline.t[i+1][link][1]

    CONNECT pipeline.t[number.of.transputers-1][link][3] TO
            pipeline.t[0][link][3]

    CONNECT pipeline.t[0][link][1] TO HOST WITH Hostlink
:

--mapping
VAL number.of.elements IS string.length:
--number.of.elements/number.of.transputers must be >= 2
VAL elements.per.transputer IS number.of.elements/number.of.transputers:
VAL remaining.elements IS number.of.elements\number.of.transputers:
VAL elements.on.root IS elements.per.transputer + remaining.elements:
NODE inout.p:
[number.of.elements]NODE pipe.element.p:

MAPPING
  DO
    MAP inout.p ONTO pipeline.t[0] PRI HIGH

    DO i = 0 FOR elements.on.root-1
      MAP pipe.element.p[i] ONTO pipeline.t[0] PRI LOW

    MAP pipe.element.p[elements.on.root-1] ONTO pipeline.t[0] PRI HIGH

    DO j = 0 FOR number.of.transputers - 1
      VAL first.element.here IS elements.on.root +(j*elements.per.transputer):
      VAL last.element.here IS first.element.here +(elements.per.transputer-1):
      DO
        MAP pipe.element.p[first.element.here] ONTO pipeline.t[j+1] PRI HIGH
        DO i = first.element.here + 1 FOR elements.per.transputer - 2
         MAP pipe.element.p[i] ONTO pipeline.t[j+1] PRI LOW
        MAP pipe.element.p[last.element.here] ONTO pipeline.t[j+1] PRI HIGH
:

--software description
#INCLUDE "hostio.inc"
#INCLUDE "sorthdr.inc"
#USE "inout.lku"
#USE "element.lku"
#INCLUDE "sortsoft.inc"
```

In the mapping structure shown, the logical processors named in the software description are mapped onto the physical processors declared in the hardware description. **Note**: On each processor, processes which communicate on external channels are mapped to be run at high priority.

The allocation of processes to transputers is shown in Figure 6.7. The number of elements on each processor depends on the maximum string length permitted by the program and the number of transputers in the pipeline.

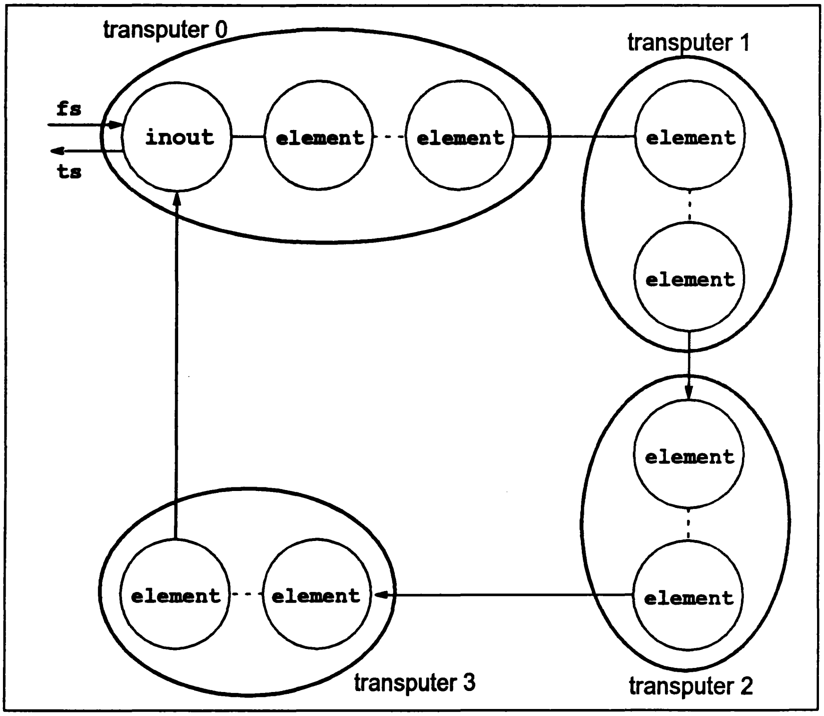


Figure 6.7 Pipeline sorter processes

6.6.1 Building the program

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

```
oc element -t425
```
(UNIX)

```
oc element /t425
```
(MS-DOS/VMS)

Because the file has a `.occ` file extension you can omit the extension from the filename. The command line option to specify the error mode may be omitted because the default is required i.e. HALT mode, but the T425 transputer target must be specified. The compiler creates a file called `element.tco`.

Next compile the input/output process:

```
oc inout -t425                              (UNIX)

oc inout /t425                              (MS-DOS/VMS)
```

This creates the file `inout.tco`.

These files must now be linked. Because the two processes are to be placed on separate processors, each must be linked individually, together with any files they reference. Each linking operation creates a unit of code which may be loaded onto the transputer network, according to the configuration defined in the configuration description.

To link `element.tco`:

```
ilink element.tco -f occama.lnk -t425       (UNIX)

ilink element.tco /f occama.lnk /t425       (MS-DOS/VMS)
```

This creates a file called `element.lku`. The linker indirect file `occama.lnk` contains the necessary references to the compiler libraries. (This file is supplied with the toolset.)

To link `inout.tco`:

```
ilink inout.tco hostio.lib -f occama.lnk -t425
(UNIX)

ilink inout.tco hostio.lib /f occama.lnk /t425
(MS-DOS/VMS)
```

This creates a file called `inout.lku`.

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

```
occonf sortconf
```

This creates an output file called `sortconf.cfb`. The input file extension can be omitted as `occonf` assumes `.pgm`.

Finally the program must be made executable. To do this run the collector tool `icollect` on the `.cfb` file.

```
icollect sortconf.cfb
```

This creates the bootable file `sortconf.btl` is created. The file extension is required.

6.6.2 Running the program

Load and run the bootable file on the transputer network using **iserver**:

```
iserver -se -sb sortconf.btl                (UNIX)

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

The '**se**' option directs the server to terminate if the program sets the error flag.

If the pipeline network is connected to the host via a root transputer use the skip loader to jump over the root transputer, and use the **iserver** '**ss**' and '**sc**' options rather than '**sb**'. In the following example the pipeline network is connected to link 2 of the root transputer:

```
iskip 2 -e -r
iserver -se -ss -sc sortconf.btl            (UNIX)

iskip 2 /e /r
iserver /se /ss /sc sortconf.btl            (MS-DOS/VMS)
```

In either case the program sorts each line of input until terminated by a blank line.

6.6.3 Automated program building

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

Note: **sortmak.pgm** compiles the program for transputer class TA in HALT error mode – it references the linked units as **.cah** files and is configured for T425 transputers. For a list of transputer targets see appendix B in the *occam 2 Toolset Reference Manual.*

To produce a Makefile for the program type:

```
imakef sortmak.btl
```

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

To build the program run your MAKE program on the file **sortmak.mak** and all the necessary compiling, linking and configuration will be done automatically. For more information about MAKE programs see Chapter 11 in the *occam 2 Toolset Reference Manual.*

6.6.4 Other configuration examples

Example **.pgm** files which configure the sorter program for other networks are supplied on the sorter directory. Descriptions can be found in the source files and in the readme file for the directory.

6.7 Conditional configurations

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

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

6.7.1 Example: Configuration using conditional `IF`

In this example, when a single processor is in use, the application communicates directly with the host, as shown in Figure 6.8. When two processors are available, a buffer process is loaded onto the root processor. This process buffers the communication between the application and the host. See Figure 6.9.

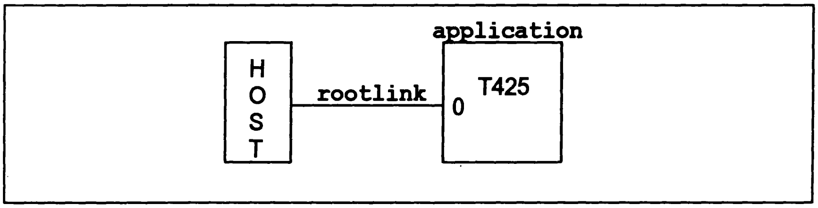


Figure 6.8 Direct host connection

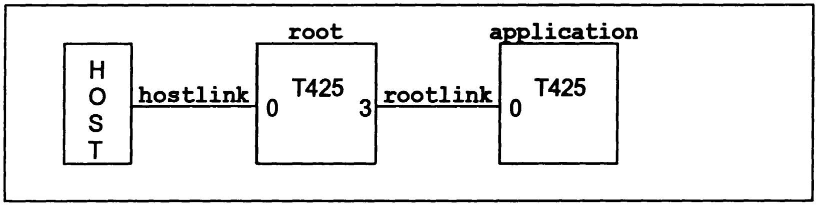


Figure 6.9 Communication via the root processor

The implementation is split into the following files:

`app.occ`	– the application
`buff.occ`	– the buffer process
`myprog.pgm`	– the configuration description file

The content of `app.occ` is as follows:

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

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

The content of `buff.occ` is as follows:

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

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

The content of `myprog.pgm` is as follows:

```
VAL number.of.processors IS 1 :  -- 1 when running,
                                 -- 2 for developing
NODE root, application :
ARC hostlink, rootlink :

NETWORK
  DO
    IF
      number.of.processors = 2
        DO
          SET root (type, memsize := "T425", #100000)
          CONNECT root[link][0] TO HOST WITH hostlink
          CONNECT root[link][3] TO application[link][0] WITH rootlink
      TRUE
        CONNECT application[link][0] TO HOST WITH rootlink
    SET application(type, memsize := "T414", #100000)
:

#INCLUDE "hostio.inc"
#USE "app.cah"
#USE "buff.cah"
CONFIG
  CHAN OF SP fs, ts :
  PLACE fs, ts ON rootlink : -- Note that this is 'rootlink', not
                             -- 'hostlink'
  PAR
    IF
      number.of.processors = 2
        CHAN OF SP fs0, ts0 :
        PLACE fs0, ts0 ON hostlink :
        PROCESSOR root
          buffer.process(fs0, ts0, ts, fs)
      TRUE
        SKIP
    PROCESSOR application
      application.process(fs, ts)
:
```

The configuration uses a constant to set the number of available processors. This is then used to conditionally build the program for one or two transputers. `NODE`s which are declared, but do not have any attributes set, are ignored when configuring a program.

The program can be built manually or using `imakef`. **Note:** when building the program for two processors, warning messages will be generated concerning interactive debugging; these can safely be ignored. Disabling interactive debugging with the `imakef` 'y' option will prevent the warnings being generated.

Source files can be found on the `examples/manuals/config` directory. When building manually remember to use `imakef` naming conventions – the program is configured for a T414 transputer and HALT error mode. The output of the program is 'Hello World'.

6.8 Summary of configuration steps

To summarize, 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:
3. Describing your hardware network.
4. Inserting `PROCESSOR` statements into your program and adding any necessary mapping description.
5. Compile all the separate compilation procedures that form the code for each transputer in a bottom up fashion.
6. Link each configuration procedure with its component parts into a file with the name used in `#USE` directives in the configuration source file.
7. Run the configurer on the configuration description file.
8. Collect the code using `icollect`.
9. Load the program into the network using the host file server.

Steps 5 to 8 can be automated by using `imakef` and a suitable MAKE program.

6.9 Further considerations

6.9.1 The effect of `occonf` on `idebug`

The use of command line options to `occonf` has a direct effect on the way in which the interactive/post mortem debugger `idebug` can be used to debug the program.

There are two main ways of using `occonf`:

- No special command line options, (the default) - this is compatible with either the interactive or postmortem debugger. However, the real time performance of the bootable produced may be significantly different to that produced by using the `Y` option, if there is a high incidence of channel communication between processors.

- With the `Y` command line option - this is compatible with the postmortem debugger only.

Important note: when virtual routing processes are used, `idebug` cannot jump down channels between adjacent processors. If this is required, the configurer '`NV`' option should be used to disable virtual routing.

Table 6.3 summarizes the use of the relevant options.

`occonf` command options	Effect
'`NV`'	Interactive and post-mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
default settings	Interactive and post-mortem debugging enabled. Virtual routing enabled and will be used even if not required, i.e. direct channel placements between processors will be ignored. *Not* possible to jump down channels between adjacent processors.
'`Y`' and '`NV`'	Post-mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`Y`'	Post-mortem debugging enabled. Virtual routing enabled and may be used. Only possible to jump down channels between adjacent processors if they are not used for virtual routing.

Table 6.3 Effect of `occonf` options on debugging

6.9.2 Reliable Channel Communications

There are a number of library routines that can be used to handle faults in the communication network. These can be used only on *direct* channels (see section 6.5.2). They must not be used on virtual channels, nor during debugging. The routines are:

```
PROC InputOrFail.t  (CHAN OF ANY c, []BYTE mess,
                     TIMER t, VAL INT time,
                     BOOL aborted)

PROC OutputOrFail.t (CHAN OF ANY c,
                     VAL []BYTE mess,
                     TIMER t, VAL INT time,
                     BOOL aborted)
```

```
PROC InputOrFail.c  (CHAN OF ANY c, []BYTE mess,
                     CHAN OF INT kill,
                     BOOL aborted)

PROC OutputOrFail.c (CHAN OF ANY c,
                     VAL []BYTE mess,
                     CHAN OF INT kill,
                     BOOL aborted)
```

The routines attempt a transfer of data on a channel. Those ending in `.t` include a timeout for failed communication, those ending in `.c` send a status message on another channel. If the communication succeeded normally, `aborted` is set to `FALSE`; if the communication was aborted (on timeout or link failure, depending on which routine was used), `aborted` is set to `TRUE`.

Note: These routines are *not* intended as the normal mode of communications. They have a higher overhead than other methods.

A further routine is available to reset a channel that has gone awry in its communications. This is:

```
PROC Reinitialise  (CHAN OF ANY c)
```

For example, when a hard link is quiescent it can be reset by this routine.

Full descriptions of all these routines can be found in section 1.10 of the *occam 2 Toolset Language and Libraries Reference Manual.*

Important note: These routines should not be used for checking the communications within a network if there is any doubt as to whether the data might not have transferred in a given amount of time. In general, you should be absolutely sure that the failure is due to a hardware failure, and not to the receiving or sending device being very busy. If the communication is terminated while data is actually being transmitted, then the results are undefined, and could stop one or both of the processors.

There is no point in using these routines on soft channels, because the communication on soft channels can be assumed to be secure.

6.9.3 Checking the configuration

Configurations may be checked against the hardware on a transputer board using a network check program such as `ispy`. The `ispy` program is supplied as part of the support software for some INMOS *i*q systems products.

INMOS *i*q systems products are available separately through your local INMOS/ SGS-THOMSON authorized distributor or SGS-THOMSON sales office.

7 Loading transputer programs

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

7.1 Introduction

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

Bootable files are generated by **`icollect`** from configuration data files (network programs) or linked units (single transputer programs). Bootable files are generated with the default extension `.btl` (for loading onto boot from link boards), or `.btr` (for loading onto boot-from-ROM boards). **Note**: a bootable file is constructed such that copying it to a link will boot the network automatically.

7.2 Tools for loading

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

- **`iserver`** — the file server and loader tool.

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

- **`iskip`** — the skip loading tool.

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

Skip loading is useful for the post-mortem debugging of programs that do not use the root transputer. The root transputer in the network is omitted from the logical network and the program is loaded onto the first processor *after* the root transputer, leaving it free to run the debugger. This avoids having to debug the code from a memory dump file.

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

7.3 The boot from link loading mechanism

`iserver` loads programs onto transputer networks, via the host link connection. It does this by simply copying the contents of the bootable file to the link. The bootable file contains all the bootstrap and loader code to ensure that the program is loaded onto the network and starts running.

The server has to be told which link connection to use and how to access it. This is done by specifying the name of a User Link on the command line or in the environment variable `TRANSPUTER`. The server gets information about the specified User Link from a connection database file. See the `iserver` documentation in chapter 13 of the *Toolset Reference Manual*.

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

When all of the code is loaded into the transputer's memory, the program starts running and can communicate with the host using the standard library routines for input and output. The libraries actually communicate with the host via the server using a predefined communication protocol known as the '`SP`' protocol. This protocol is defined in the `iserver` documentation.

The program continues to run until: an error occurs, the server is terminated by pressing the `iserver` interrupt key (usually CTRL-C or CTRL-BREAK), or the program terminates naturally. **Note**: terminating the server will not stop the program running on the transputer. However, any processes on the transputer which attempt to communicate with the server will deadlock. This may eventually cause the whole program to stop as other processes become dependent on this communication. The program may be able to continue if the server is restarted.

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

7.4 Boards and subnetworks

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

Boot from link TRAM boards form the majority of transputer boards in general use. They are loaded down the link that connects the root transputer to the host using the `iserver` tool. Programs intended to run on boot from link boards must consist of bootable code, such as that generated by `icollect`.

Examples of boot from link boards supplied by INMOS are the IMS B008 PC motherboard (with appropriate TRAMs) and the IMS B014and IMS B016 VME bus standard interface boards.

Boot from ROM TRAM boards are intended for stand-alone applications such as embedded systems.

7.4.1 Subsystem wiring

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

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

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

The Down and Subsystem ports work in the following ways:

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

Subsystem propagates the **Reset** and **Analyse** signals but also allows control by the board, enabling subnetworks downstream of the board to be independently reset, analyzed, and their error flags read, under the control of the transputer to which the subsystem is attached.

7.4.2 Connecting subnetworks

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

In a typical multitransputer system the root transputer's Up port is connected to the host computer so that the host can control the loading of programs and monitor errors on the network. The first processor in the subnetwork is connected to either Down or Subsystem depending on the application, and other processors on the network are chained together via their Up and Down ports.

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

A more complicated application may require several programs to be loaded onto the subnetwork under the control of the root transputer. Here the subnetwork would be connected to Subsystem so that the root transputer could repeatedly reset and re-load the subnetwork. Any errors in the subnetwork would be detected by the root transputer through its Subsystem port, and the error would not be propagated through the Up port to the host computer. **Reset** and **Analyse** signals are propagated through to the Subsystem port, but the error signal is not relayed back. (**Note** some boards do not conform to this system of signal propagation, see 7.5.2).

7.5 Loading programs for debugging

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

Commands to use for various combinations of subsystem wiring, board type, and program mode, are listed in the debugger reference documentation.

7.5.1 Breakpoint debugging

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

7.5.2 Board types

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

7.5.3 Use of the root transputer

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

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

1 Programs can be loaded in the normal way using `iserver`, and the program image in the root transputer's memory saved to a file. The code running on the root transputer is then debugged from the dump file. Code running on the rest of the network is debugged in the normal way by reading the transputer memory directly down the transputer links. The dump file is created by invoking `idump`. The debugger is subsequently invoked using the debugger '`R`' option that directs it to read the dump file.

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

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

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

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

Details of the procedures to use for loading and debugging all types of transputer programs can be found in section 4.2 of the *Toolset Reference Manual.*

7.5.4 Analyse and Reset

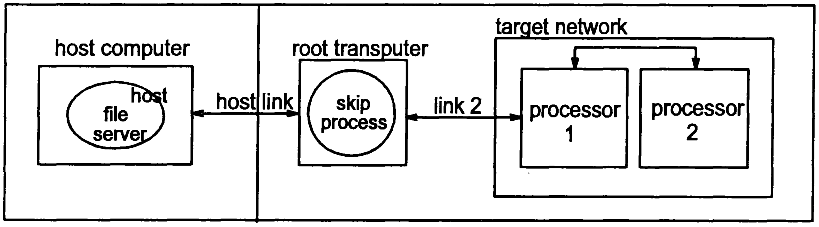
Care must be taken that **Analyse** or **Reset** are only asserted once on a network that is to be debugged, or incorrect data will be obtained. To ensure this the debugger should be invoked using the standard command sequences given in the debugger reference documentation.

7.6 Example skip load

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

7.6.1 Target network

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



7.6.2 Loading the program

Assume the file `twinprog.btl` contains the bootable program. To prepare the board for running the program on the target network, first invoke `iskip`:

```
iskip 2 -r -e                                   (UNIX)

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

This command sets up link 2 as the path to the target network and starts the route-through process on the root transputer. Options '`r`' and '`e`' respectively reset the target network and direct the host file server to monitor the halt-on-error flag. The program can then be loaded:

```
iserver -ss -se -sc twinprog.btl                (UNIX)

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

Note: these examples assume that the environment variable `TRANSPUTER` has been defined to specify the name of the User Link to use to access the transputer network, and that a connection database file exists to define that User Link. See the `iserver` documentation (chapter 13 of the *Toolset Reference Manual*) for more details. Options '`ss`' and '`sc`' are used in place of '`sb`' because the network has already been reset by `iskip`.

See chapter 15 of the *Toolset Reference Manual* for more information on the `iskip` tool.

7.6.3 Clearing the network

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

The `ispy` program is provided as part of the support software for some INMOS *iq* systems products. These products are available separately through your local INMOS distributor or SGS-THOMSON Sales Office.

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

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 5.

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 C of the *occam 2 Toolset Reference Manual.*

8.2.2 Library support

Two i/o libraries are provided for accessing the file system and other host services. The libraries are summarized below.

`hostio.lib`	File and terminal i/o; host access
`streamio.lib`	Stream-based terminal and file i/o

All routines in these libraries are independent of the host operating system.

The hostio library contains basic routines for accessing files and controlling the file system. It also contains routines for general interaction with the host. Use the hostio library for basic file operations, and for accessing host services.

The streamio library contains routines for creating and outputting to streams. It also provides primitives for reading and writing text and numbers, and for controlling the screen. Use the streamio library for inputting and outputting character and data streams.

Definitions of constants and protocols used within the libraries are provided in the include files `hostio.inc` and `streamio.inc`. These files should be included in all programs where the respective libraries are used.

Details of all i/o procedures and functions can be found in the *occam 2 Toolset Language and Libraries Reference Manual.*

8.2.3 File streams

The host file server supports a stream model of file and terminal access. When a file is opened a 32-bit integer stream id is returned to the program. This identifier must be quoted by the program whenever the file is accessed, and is valid until the file is closed. Streams and files must be explicitly closed by the programs that use them, and the server must be explicitly terminated when the program finishes and host services are no longer required.

Three streams are predefined:

`spid.stdin`	standard input
`spid.stdout`	standard output
`spid.stderr`	standard error

These streams can be closed by the programmer, but cannot be reopened. Take care not to close the standard streams if you are using hostio routines that read or write to them. The streams can only be closed by specifying the streamid explicitly and cannot be closed inadvertently using the hostio routines.

Standard input and output are normally connected to the keyboard and screen respectively, but may be redirected by the operating system. Streams and files other than the three standard streams described above must be explicitly closed by the program. When the program finishes and host services are no longer required, the server should be terminated by the transputer application calling `so.exit`.

Protocols

occam programs communicate with the host file server through a pair of occam channels. Requests for service are sent to the host on one channel and replies are received on the other. Both channels use the `SP` protocol, which is defined in the include file `hostio.inc`.

8.3 Host implementation differences

The IBM PC version of the host file server supports a number of DOS specific commands via routines in the library file `msdos.lib`. The VAX/VMS and UNIX implementations have no host specific commands.

If you wish to write programs that are portable between all implementations of the toolset you are recommended to use only host independent routines. All procedures and functions in the hostio and streamio libraries are host independent.

8.4 Accessing the host from a program

For programs to be run on transputer boards the host is accessed through the channels `fs` and `ts`, both defined as `CHAN OF SP`. Protocol `SP` is defined in the include file `hostio.inc`. For single transputer programs the channels are defined within the program, and for multiprocessor programs the channels are placed on the link that is connected to the host. The normal location for the connection to the host is link zero on the root processor.

8.4.1 Using the simulator

The simulator tool `isim` provides access to the host file server in the same way as a single processor program running on a board, connected via link 0.

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.

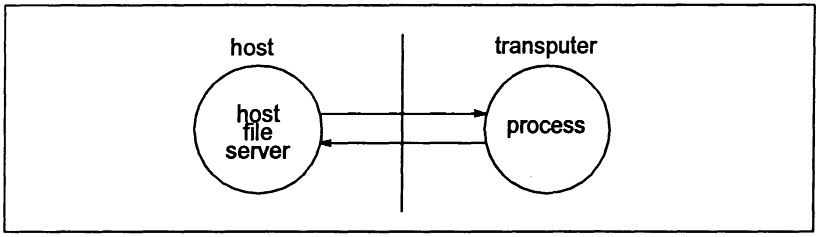


Figure 8.1 Program input/output

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.

Four 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: `so.multiplexor`; `so.overlapped.multiplexor`; `so.pri.multiplexor`; and `so.overlapped.pri.multiplexor`. Details of the routines can be found in section 1.5.9 of the *occam 2 Toolset Language and Libraries Reference Manual.*

An example of a multiplexed system is shown in Figure 8.2, and the occam code that would implement the system is listed in Figure 8.3.

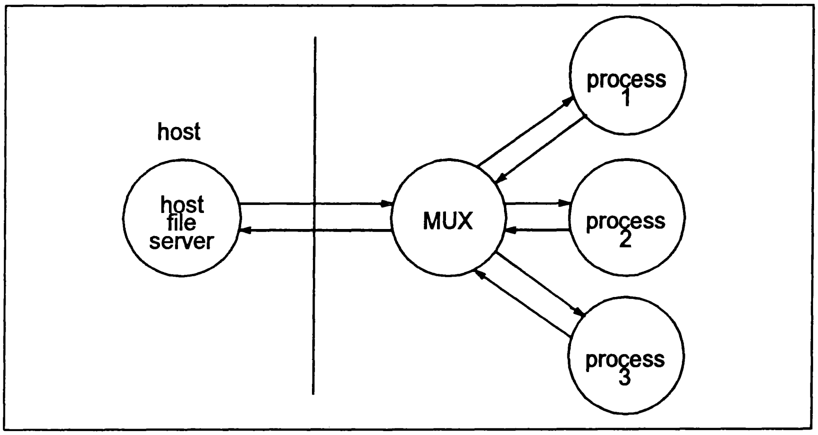


Figure 8.2 Multiplexing the host file server

```
#INCLUDE "hostio.inc"  -- SP protocol declaration
PROC mux.example (CHAN OF SP fs, ts,
                  []INT free.memory)

  #USE "hostio.lib"  -- host i/o libraries

  #USE "process0"    -- user processes
  #USE "process1"
  #USE "process2"
  SEQ
    CHAN OF BOOL stop:
    [3]CHAN OF SP from.process, to.process:
    PAR
      so.multiplexor(fs, ts, -- server channels
                     from.process, to.process,
                        -- multiplexed channels
                     stop)   -- termination channel

      SEQ
        PAR    -- run user processes in parallel
               -- sharing the iserver
          process0(to.process[0], from.process[0])
          process1(to.process[1], from.process[1])
          process2(to.process[2], from.process[2])
        stop ! FALSE  -- terminate multiplexor
    so.exit(fs, ts, sps.success)
:
```

Figure 8.3 Multiplexing example

This source for this program can be found in `examples/manuals/mux`.

Multiplexor processes can be chained together to produce any degree of multiplexing to the host. However, the host is a single, finite resource and unrestrained multiplexing of processes should be avoided if possible.

8.5.1 Buffering processes to the host

It may sometimes be useful to pass data invisibly through another process, for example when passing data to the server through intervening processes. The hostio library routine `so.buffer` takes a pair of input and output channels and passes data through unchanged.

8.5.2 Pipelining

If data has to pass through many processes before reaching the server efficiency may be improved by allowing a data transfer to begin before the previous one has completed its journey down the line of processes. This allows several data transfers to be in progress simultaneously and is known as pipelining.

The routine **so.overlapped.buffer** can pipeline several buffers up to a user-defined limit. A pipelined version of the multiplexor process called **so.overlapped.multiplexor** performs the same function for multiplexed processes. Prioritized versions of the routines may also be used.

9 Debugging transputer programs

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

Worked examples are given at the end of this chapter, the sources of which may be found in the toolset examples subdirectory.

Chapter 4 of the accompanying *Toolset Reference Manual* provides detailed information about `idebug`, including command line syntax and full descriptions of the symbolic debugging and monitor page commands.

9.1 Introduction

The network debugger `idebug` is a symbolic debugger for transputers and transputer networks. It can be used to examine stopped programs (post-mortem debugging) or to debug programs interactively (breakpoint debugging). It can be used with INMOS ANSI C, occam, and FORTRAN-77 programs, and with mixed language systems.

Programs can be analyzed using the *symbolic* functions which operate using source code symbols or the *monitor page* commands which operate at memory and processor level.

Symbolic functions allow files to be examined, variables inspected, and procedures traced, from source code level. Monitor page commands allow transputer memory to be examined and processor state to be determined anywhere on the network. Symbolic and monitor page environments are separate but can be switched between at will.

`idebug` can be used to debug mixed language programs, although certain facilities are available for some languages and not others. For example, a comprehensive expression language exists for C and a simpler one for occam. The exact usage of some facilities may also differ slightly between languages.

9.1.1 Post-mortem debugging

Post-mortem mode debugging allows stopped programs to be analyzed from the residual contents of transputer memory or from a network dump file. Programs that run on the root transputer must be debugged from a memory dump file because the debugger overwrites the root transputer's memory. The memory dump file is created using the `idump` tool (see chapters 4 and 5 of the accompanying *Toolset Reference Manual*).

9.1.2 Interactive debugging

Interactive debugging (also known as *breakpoint mode* debugging) allows transputer programs to be executed interactively using breakpoints set in the code.

Breakpoints can be set symbolically on lines of source text or at transputer memory addresses, and values can be modified in transputer memory to show the effect of changing variables. Breakpoint mode debugging requires the use of two or more transputers, because the debugger tool runs on the root transputer.

Certain symbolic functions and monitor page commands are only available in breakpoint debugging mode.

9.1.3 Mixed language debugging

When debugging programs constructed from a mixture of languages from different INMOS toolsets, you should always use the version of `idebug` with the highest version number (as displayed in the help or monitor pages). This is true for all versions of `idebug` with a version number greater than `V2.00.00`. This will help ensure that no toolset incompatibilities occur.

9.1.4 Debugging with `isim`

The transputer simulator tool `isim` can also be used to debug transputer programs from a low level environment. Using a similar environment to the debugger monitor page transputer memory can be examined, breakpoints set, and programs executed by single stepping.

The debugging facilities of the simulator are briefly described in this chapter (section 9.13). Details of how to use the simulator tool can be found in chapter 14 of the accompanying *Toolset Reference Manual*.

9.2 Programs that can be debugged

The debugger can analyze 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 *root* transputer is the processor that is directly connected to the host computer. In a transputer network that is connected to the host it forms the root of the network. The debugger always runs on the root transputer, which must be a 32-bit transputer with at least one megabyte of memory (preferably two or more).

The relationship of the root transputer to the host computer and the rest of the network is illustrated in figure 9.1.

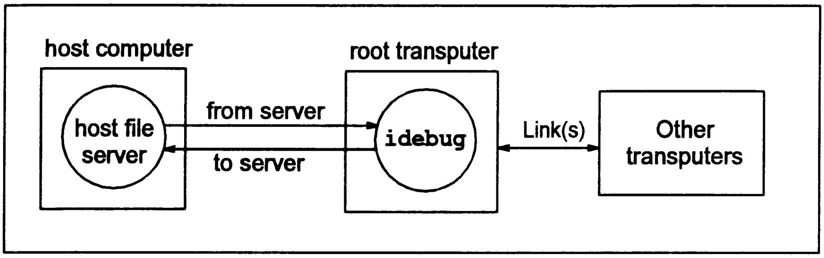


Figure 9.1 Debugging a transputer network

If breakpoint debugging is used the transputer network must contain at least two processors because the root transputer is dedicated to running the breakpoint debugger in parallel with the user's program.

9.3 Compiling programs for debugging

Programs to be debugged should be compiled with full symbolic debugging information enabled. For C and FORTRAN, this is achieved by specifying the compiler 'G' option when the program is compiled. The occam compiler generates object files containing full debugging information by default. Two command line options may be used to limit the debugging information produced by the compiler.

Minimal debugging information

By default the C and FORTRAN compilers generate object files containing minimal symbolic debug information so that object modules, especially those to be used as libraries, are kept as small as possible. Minimal debug information enables the debugger to backtrace out of a library function to a module compiled with full debug information.

occam programs can be compiled with minimal debug information by using the compiler 'D' option.

Note: The object code produced by the C and FORTRAN compilers with minimal debug information contains certain optimizations that are absent in code generated with full debugging information enabled. As a consequence the object code produced may differ slightly from code compiled with full debugging information enabled.

occam channel communication

The 'Y' option to the occam compiler disables channel communication via library routines and, instead, produces optimal in-line code for channel i/o. Interactive debugging requires all communications to be done by means of the library routines, so this option also disables interactive debugging.

C channel communication

Use of the C library `DirectChan` functions on channels provided by the configurer will interfere with and corrupt interactive debugging. Note that the `DirectChan` functions can be safely used with edges passed from the configurer, and with internal (soft) channels declared in C source files.

9.3.1 Error modes

Programs to be debugged should be generated in HALT mode, which is the linker default. The behavior of a program when an error occurs depends on the mode in which the program was compiled and linked, as follows:

- In HALT mode any error during program execution halts the transputer immediately.
- In STOP mode, errors do not halt the program, rather they stop the errant process allowing other processes executing on the same transputer to continue. Programs compiled in this mode can only be debugged if they are halted explicitly.
- Programs compiled in UNIVERSAL mode will adopt the error mode selected at link time i.e. HALT or STOP mode. If UNIVERSAL mode is selected at both compile and link time, then the error behavior will default to HALT mode.

By default, C and FORTRAN programs are compiled in UNIVERSAL error mode and linked in HALT mode. By default, occam programs are compiled in HALT mode and linked in HALT mode.

9.4 Debugging configured programs

Programs configured with the C-style configurer, `icconf`, must have debugging enabled by means of the appropriate `icconf` command line options. occam programs are compiled, linked, and configured with interactive debugging enabled by default. Debugging can be disabled in occam modules by the appropriate `occonf` command line options.

Table 9.1 summarizes the effects of the relevant `icconf` and `occonf` options on interactive and post-mortem debugging, and on virtual routing.

`icconf` command options	`occonf` command options	Effect
'`g`' and '`nv`'	'`nv`'	Interactive and post-mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`g`'	*default settings*	Interactive and post-mortem debugging enabled. Virtual routing enabled and *will* be used even if not required, i.e. direct channel placements between processors will be ignored. *Not* possible to jump down channels between adjacent processors.
'`gp`' and '`nv`'	'`y`' and '`nv`'	Post-mortem debugging enabled. Virtual routing disabled. Possible to jump down channels between adjacent processors.
'`gp`'	'`y`'	Post-mortem debugging enabled. Virtual routing enabled and *may* be used. Only possible to jump down channels between adjacent processors if they are *not* used for virtual routing.
Note: the `icconf` '`gp`' and the `occonf` '`y`' options are not equivalent. For further details of configuration options, see the *Toolset Reference Manual*.		

Table 9.1 Effect of `icconf` and `occonf` options on debugging

9.4.1 Debugging with configuration level channels

`idebug` cannot locate to a process waiting on a transputer link, or locate to a process (on a different processor) waiting on a channel mapped onto a link, if that link is used by the configurer for software virtual channels.

`idebug` *is* able to locate to a process waiting on a transputer link or jump down a channel between two processes (which may be on different processors) if the channel is one of the following:

- An internal (soft) channel between processes on the same processor.
- An external (hard) channel between processes on different processors which is not used by the configurer for software virtual links.

9.4.2 Debugging with the configurer reserved attribute

The reserved attribute should *not* be specified to the configurer in order to reserve on-chip memory if you wish to interactively breakpoint debug the program. This is because the runtime kernel (see section 9.7.1) which the debugger places on each processor reserves the first 11K – 15K of memory for its own use (regardless of the `reserved` attribute being specified to the configurer).

9.5 Debugging boot from ROM programs

Programs configured using the `icconf` '`GP`' option or the `occonf` '`Y`' option (see table 9.1) may also be debugged in boot from ROM run in RAM systems (configurer '`RA`' option).

9.6 Post-mortem debugging

Post-mortem debugging is the analysis of stopped programs, that is, programs that have failed to run correctly and have set the transputer error flag (or have detected a *hard* parity error). Programs that are to be debugged in this mode should be compiled and linked in HALT mode (HALT is the linker default) so that the processor halts when the flag is set. They should be loaded by `iserver` using the '`SE`' option, so that the error flag is monitored and the server terminated if the error flag is set.

The conditions in which the transputer error flag may be set depend on the language being used. C and FORTRAN provide little or no automatic checking of errors whereas occam provides comprehensive error checking by default.

C programs can also set the error flag and halt the processor when the program is terminated by functions such as `halt_processor`, `abort`, `assert`, `debug_stop` or `debug_assert`.

9.6.1 C and FORTRAN programs

Little automatic error checking is provided in C or FORTRAN — this can make it difficult to cause a program to halt when an error occurs. This rather restricts the usefulness of post-mortem debugging, but it can be used if programs are halted explicitly using the debugging support functions such as `debug_assert()` (`DEBUG_ASSERT()` in FORTRAN) etc. These functions are described more fully in the appropriate *Language and Libraries Reference Manual* and in the debugging examples.

Breakpoint debugging, with its associated debugging support functions, is a more flexible approach and is the recommended method when debugging C and FORTRAN programs.

The C library `abort()` function can be enabled to halt the processor by calling the auxiliary function `set_abort_action()`. This enables a backtrace to be performed to the point in a program where the error occurred without the need to modify any of the `assert()` statements contained in the program.

This technique is illustrated with the following example (which is contained in the C toolset debugger examples directory):

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

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

int
main (void)

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

        printf ("Program started\n");

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

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

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

        exit (EXIT_SUCCESS);
}
```

9.6.2 occam programs

The runtime errors that can cause an occam program to set the error flag and halt include:

- An arithmetic error, such as overflow or divide by zero, has occurred.
- An array index is out of range.
- A value is out of range in a type conversion.
- An alignment error has occurred in a type conversion or abbreviation
- An array element is being 'aliased' at run-time — that is, being referred to by more than one name within a given scope.
- A `STOP` process, or a process which behaves like `STOP` (e.g. an `IF` with no `TRUE` guards, or an `ALT` with no enabled guards), being executed.

When a run-time error occurs, the program halts the processor and allows the debugger to enter the program for post-mortem debugging.

In addition, some debug support functions (e.g. `DEBUG.ASSERT()`) are provided to aid debugging of programs by implementing an explicit program error; details of these functions can be found in the *occam 2 Toolset Language and Libraries Reference Manual* and in the debugging examples.

9.6.3 Interrupted programs

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

9.6.4 Parity errors

The T426 will detect two types of parity errors, hard and soft. A soft error is one which disappears on retry; it does not stop the processor but sets, and resets, the **SoftParityError** pin. This allows soft errors to be monitored externally (or internally if **SoftParityError** is connected back to the **Event** input). A hard error occurs if a location still causes a parity error on retry; in this case the processor is stopped immediately and the **HardParityError** pin is asserted.

After a hard parity error has been detected the debugger can be started in post-mortem mode. If the debugger fails to find a processor which has halted with the error flag set, it will try to find a T426 processor which has had a hard parity error. It will then display this as the first processor in error. The debugger does not automatically locate to the program source if a parity error has occurred — the debugger will instead display the monitor page to allow the parity registers to be examined.

The parity registers are displayed on the monitor page at the bottom left of the display below the clock registers. These registers are *not* displayed in interactive mode. This is because the registers are volatile and reading the registers would interfere with any user code attempting to handle soft parity errors.

9.6.5 Debugging the root transputer

Programs which run on the root transputer, or which use the root transputer to run part of a multiprocessor program, must be debugged 'off-line' from a separately created memory image file. This is necessary because the debugger executes on the root transputer and overwrites the code in its memory. The memory dump is performed using the `idump` tool after the program has failed and before the debugger is started with the '`R`' option. Details of how to use the `idump` tool can be found in chapter 5 of the accompanying *Toolset Reference Manual.*

Skip loading

As an alternative to using the `idump` tool, the application program can be *skip* loaded onto the next processor on the network, avoiding the root transputer. This leaves the root transputer free to run the debugger. A disadvantage of this method is that it requires one extra processor on the network in addition to those needed for the program.

If only one transputer is available, for example on single-transputer boards, the memory dump method *must* be used. If more than one transputer is available skip loading is the recommended method since it enables the program to be directly debugged from transputer memory.

Use of the skip loader is described in chapter 7 of this manual and chapter 15 of the accompanying *Toolset Reference Manual.*

9.7 Interactive debugging

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

Programs that communicate to the host *must* use `iserver` protocol, as used by the standard I/O libraries, when being debugged interactively.

9.7.1 Runtime kernel

The breakpoint debugger places a special runtime kernel on each processor in addition to the application bootable code. This kernel provides a communication network to enable the debugger to transparently share transputer links with the application in addition to providing a breakpoint handler to deal with breakpoints, errors, inspection of processor state etc. The scheme is illustrated in Figure 9.2.

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

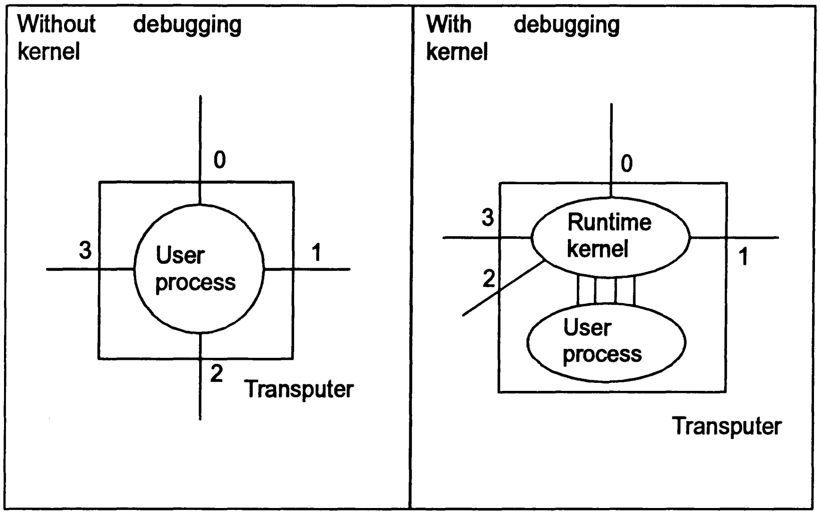


Figure 9.2 Debugger runtime kernel

The runtime kernel requires a certain amount of memory on each processor, the exact amount differing slightly between processor types. Kernels on processors with hardware support require slightly more memory because they retain more state information. The size of the kernel on each transputer type is given in table 9.2.

Apart from the extra memory required, the kernel is transparent to the application program if processes on different processors communicate with each other in the normal way, using channels supplied by the configurer.

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

Processor	Kernel size	H/W support
M212	11.25K	No
T212	11.25K	No
T222	11.25K	No
T225	12.75K	Yes
T414	13.5K	No
T800	13.5K	No
T400	15.25K	Yes
T425	15.25K	Yes
T426	15.25K	Yes
T801	15.25K	Yes
T805	15.25K	Yes

Table 9.2 Runtime kernel size and processor breakpoint support

9.7.2 Processors without hardware breakpoint support

Certain transputers have built-in instructions to aid breakpointing (see table 9.2). For those processors without hardware breakpoint support, breakpoints should not be set within *high priority* processes because the mechanism used to implement breakpoints causes such processes to lock the processor and disables all communications to the processor via the runtime kernel. To help safeguard against this problem, the debugger monitor page breakpoint option will only set breakpoints at high priority process entry points or `main()` on processors with hardware breakpoint support.

The exact effect on the network of encountering such a breakpoint will depend on the position of the processor in the network hierarchy but the possibility should be avoided. Since the debugger is, in general,unable to check the validity of breakpoints it is the programmer's responsibility to ensure correct operation on processors without direct hardware breakpoint support.

9.7.3 Creating programs for debugging

Programs to be debugged using breakpoint debugging should be compiled with full debug information enabled, using the C and FORTRAN compilers '`G`' option and the occam compilers default.

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

The code must be produced *without* using the '`Y`' option with any of the tools if interactive debugging is to be done.

9.7.4 Loading the program

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

Clearing error flags

If either `iserver` or `idebug` detect that the error flag is set immediately a program begins to run, it is likely that the network contains more processors than you are currently using, and that one or more of the unused processors has its error flag set. The error flag may be randomly set when the transputer is powered up — it is normally cleared by the bootstrap code.

The error flags of all the processors in a network can be cleared by running a network check program such as `ispy`. This ensures a clean network on which to load the program. This generally only needs to be done once, after the system is first turned on.

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

An alternative way of clearing all the error flags in the network is to load a dummy program which is configured to use every processor in the network. In the act of loading the dummy code the processor error flag is cleared.

Parity-checked memory

In system that include some processors which have external memory with parity-checking (e.g. systems built with the T426) it is necessary to initialize the contents of memory before the application code is run. This is because a read from un-initialized memory could cause a parity error to be reported.

Normally, when not breakpoint debugging, the contents of memory are initialized by the bootstrap loader code. This is controlled by the collector `CM` option (see chapter 3 of the *Toolset Reference Manual*).

The debugger has two command line options which can be used for for memory initialization — both of these are followed by a hexadecimal number representing the pattern to be written to memory. The '`J`' option writes the given pattern to all of the data areas (stack, workspace, static, heap and vectorspace as appropriate) in each processor. The '`K`' option writes to the same areas of memory as the '`J`' option and also to the 'freespace' area.

In general, the '`J`' option should be used for configured programs and the '`K`' option for non-configured programs (i.e. programs for a single processor produced using the collector's '`T`' option). The memory initialization is performed on all processors in the network, not just T426s.

These options can also be useful for seeing where data has been written to memory. For example, they can be used to determine the size of stack or heap used by a program when it runs, or to detect data written to unexpected areas of memory. Note that the bootstrap phase of a program may use a small part of the program data and freespace areas for its own purposes, consequently the pattern of data may have some holes in it.

9.7.5 Running the debugger

The command `idebug` starts the host file server program, `iserver`, to load the debugger onto the root transputer and provide it with host services. Different options need to be given to `idebug` depending on the type of debugging being done (e.g. breakpoint or post-mortem) and the details of the transputer network being used (e.g. is the code to be debugged running on the root processor or is that transputer available for the debugger). Some basic examples are given here.

Note that the transputer network is not reset or analyzed by default so, generally, one of the `iserver` options must be specified for this (e.g. '`SR`' or '`SA`'). This is true even if the '`D`' option is being used to run the debugger without using transputers (because the processor running the debugger must be reset).

When doing breakpoint debugging, the '`SR`' option is used to cause the `iserver` to reset the transputer network and the '`B`' option to specify which link from the root transputer is connected to the processors running the application code — for example:

```
idebug -sr -b 2 program.btl                 (UNIX)
idebug /sr /b 2 program.btl                 (MS-DOS/VMS)
```

As another example, when using the debugger in post-mortem mode to debug a program which does not use the root transputer, the '`SA`' option would be used to make the server put the network into **Analyse** mode with the '`T`' option to specify the link from the root processor to the transputer running the program to be debugged:

```
idebug -sa -t 2 program.btl                 (UNIX)
idebug /sa /t 2 program.btl                 (MS-DOS/VMS)
```

Finally, when debugging a program running on the root transputer in post-mortem mode, the `idump` command is first used to create a file containing a dump of the transputer's memory and then `idebug` is run with the '`R`' option to specify the core dump filename — for example:

```
idump core.dmp #100000

idebug -r core.dmp program.btl              (UNIX)
idebug /r core.dmp program.btl              (MS-DOS/VMS)
```

Complete details of which options to use in different circumstances are given in the accompanying *Toolset Reference Manual*, chapter 4.

9.7.6 Interactive mode functions and commands

Several symbolic debugging functions and monitor page commands are only available in interactive mode. The commands available are summarized below.

Symbolic functions

TOGGLE BREAK	Set or clear a breakpoint on the current line.
RESUME	Restart a process stopped at a breakpoint.
CONTINUE FROM	Restart a stopped process from the current line.
MODIFY	Change the value of a variable in memory.

Monitor page commands

B	Breakpoint menu.
J	Execute program.
S	Show debugging messages.
U	Update register display.
W	Write to memory.

9.7.7 Breakpoints

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

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

Note: When a process is stopped at a breakpoint or by one of the debugging functions (e.g. `debug_stop`) other parallel processes in the program continue to run. A side effect of pausing is that the debugger suspends `iserver` communications in order to preserve the debugger's screen display.

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

Setting breakpoints at symbolic level is the recommended method.

9.8 Program termination

Program termination is signalled to the debugger by the termination of `iserver`. This is performed automatically by the C and FORTRAN runtime systems, and must be done explicitly by the user in occam code. If the program contains independently executing processes which do not require communication with the server the debugger may be resumed to interact with these processes.

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

9.9 Symbolic facilities

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

Source level debugging is accessed through symbolic *functions* mapped to specific keyboard function keys (e.g. INSPECT) by an 'ITERM' file. Keyboard layouts for specific terminal types can be found in the Delivery Manual that accompanies this release. Alternatively, the comments in the ITERM file can be read to find the mapping of functions to keys.

Help screen

A help page can be displayed by pressing either ? or HELP, this displays the following information:

```
                        idebug Symbolic Help Summary
                        ****************************

1-INSPECT 2-CHAN 3-TOP 4-RETRACE 5-RELOC 6-INFO 7-MOD 8-RESUME 9-MONITOR 0-BACK

The above list summarises the commonly used functions available in symbolic
mode.  For a complete list of all symbolic functions available please refer to
the idebug documentation.  The mapping of a symbolic function to a particular
key may be found in the file defined by the ITERM environment variable.

INSPECT    -  Display the type and value of a variable
CHANNEL    -  Locate the process waiting on a channel
TOP        -  Locate back to the error or last source code location
RETRACE    -  Undo a BACKTRACE
RELOCATE   -  Locate back to the last location line
INFO       -  Display process information (eg. Iptr, Wdesc, process name)
MODIFY     -  Change the value of a variable in memory
RESUME     -  Resume a process stopped at a breakpoint
MONITOR    -  Change back to the Monitor page
BACKTRACE  -  Locate to the calling function or procedure
HELP or ?  -  This help summary

======================  Hit any key to continue  ======================
```

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

9.9.1 Locating to source code

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

When first started in post-mortem mode, the debugger determines the address of the last instruction executed, which it uses to automatically locate to the relevant source code. Subsequently for each new point to locate to in the code the debugger requires a new address which can be supplied by the programmer.

Process addresses can be determined using the monitor page [R], [T], and [L] commands that display the processes waiting on the run queues, the timer queues, and the transputer links. To locate to a process displayed by one of these commands, use the [G] command. Code corresponding to any memory address can be located using the monitor page [O] command.

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

The symbolic functions that can be used directly for locating to specific locations and sections of source code are listed below.

[RELOCATE]	Locate back to the last location line.
[TOP]	Locate back to the error or last source code location.
[CHANNEL]	Locate to the process waiting on a channel.

The [CHANNEL] function is described more fully in section 9.9.4.

Other functions which locate to specific sections of code are the [BACKTRACE] and [RETRACE] functions. These are used to trace subprogram calls and do not require a specified address. The functions are described in section 9.9.4.

A strategy for locating processes in multi-process programs is presented in section 9.11.

9.9.2 Browsing source code

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

TOP OF FILE	Go to the first line.
END OF FILE	Go to the last line.
GOTO LINE	Go to a specified line.
SEARCH	Search for a specified string.
ENTER FILE	Enter an included source file.
EXIT FILE	Exit an included source file back to the enclosing source file.
CHANGE FILE	Display a different source file.

9.9.3 Inspecting source code and variables

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

INSPECT	Display the type and value of a source code symbol.
CHANNEL	Locate to the process waiting on a channel.
TOGGLE HEX	Enables/disables Hex-oriented display of constants and variables. Selects the display of source code symbols in hexadecimal form for C and FORTRAN.
GET ADDRESS	Displays the start address of the sequence of transputer instructions corresponding to the selected source line.
INFO	Displays low-level information about the selected process.

9.9.4 Jumping down channels

The CHANNEL function can be used to locate to a process waiting on a channel. This is known as 'jumping down' a channel and works for channels on the same processor (internal or *soft* channels) or channels assigned in the configuration to transputer links (external or *hard* channels which connect processes on different processors together). It *cannot* be used to jump down software virtual links provided by the configurer. Debugging can then continue at the waiting process. If no process is waiting on a channel the channel is reported as '`Empty`'.

9.9.5 Tracing procedure calls

Two functions assist in the tracing of procedure and function calls. They can be used even if the source of the called routine is not present, for example, libraries supplied as object code with minimal debug information. In this case the line containing the function call is displayed rather than the library code itself. Where procedures are nested, successive backtrace operations will locate to the original call. Variables and other symbols can be inspected at any stage. The two functions are listed below.

BACKTRACE	Locate to the calling procedure or function.
RETRACE	Undo a BACKTRACE.

9.9.6 Modifying variables

The MODIFY function allows variables to be changed in transputer memory and the program continued with the new values. For C and FORTRAN it supports the same expression language as INSPECT. For further details see chapter 4 in the *Toolset Reference Manual.*

9.9.7 Breakpointing

Symbolic functions are provided for setting and clearing breakpoints, for modifying the value of a variable, and for continuing the program.

TOGGLE BREAK	Set or clear a breakpoint on the current line.
RESUME	Restart a process stopped at a breakpoint.
CONTINUE FROM	Restart a stopped process from the current line.
INTERRUPT	Force the debugger into the monitor page (without necessarily stopping the program).
MODIFY	Change the value of a variable in memory.

9.9.8 Miscellaneous functions

The following extra functions are available at symbolic level:

MONITOR	Change to the monitor page.
FINISH	Quit the debugger.

9.10 Monitor page

The debugger monitor page is a low level debugging environment which gives direct access to machine level data. It allows memory to be viewed and disassembled and gives access to information about the processor's activity through the display of error flag status and pointers to process queues. Specific debugging operations are selected by single letter commands typed after the '`Option`' prompt.

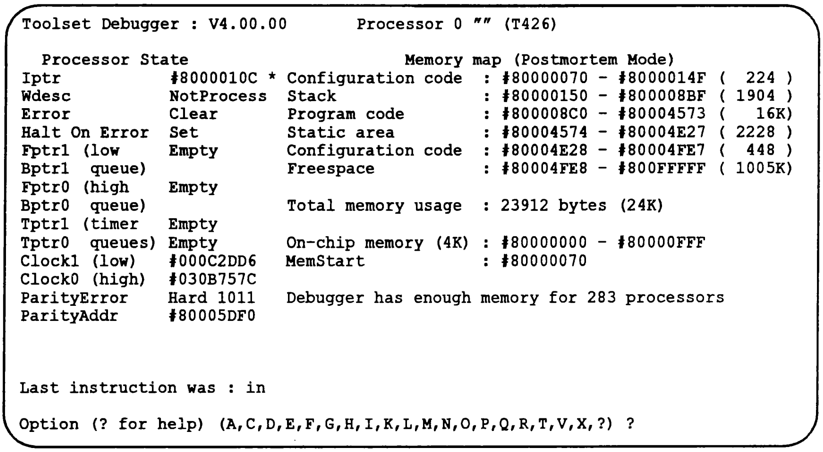
9.10.1 Startup display

When first started in interactive mode, or in post-mortem mode with an invalid **Iptr** or **Wdesc** (see below), the debugger enters the monitor page environment and displays information such as the addresses of instruction and workspace pointers, status of error flags, and information about the processor run queues. The memory map is also displayed.

If an **Iptr** or **Wdesc** is invalid at startup it is indicated by an asterisk ('*'). A double asterisk ('**') is used to indicate an **Iptr** or **Wdesc** which is outside the defined memory on a processor (i.e. beyond the 'freespace').

The monitor page display differs slightly between post-mortem and breakpoint modes. In post-mortem mode the display includes the saved pointers for the low priority process if the processor was running at high priority when analyzed; in breakpoint mode the display does not include these pointers but does include the contents of the registers **Areg**, **Breg**, and **Creg**, if known. At startup in breakpoint mode, no machine pointers or register values are available (the program has not yet started) and so no values are displayed.

A typical startup display is shown in Figure 9.3.



```
Toolset Debugger : V4.00.00       Processor 0 "" (T426)

  Processor State                         Memory map (Postmortem Mode)
Iptr           #8000010C * Configuration code  : #80000070 - #8000014F (  224 )
Wdesc          NotProcess  Stack               : #80000150 - #800008BF ( 1904 )
Error          Clear       Program code        : #800008C0 - #80004573 (   16K)
Halt On Error  Set         Static area         : #80004574 - #80004E27 ( 2228 )
Fptrl (low     Empty       Configuration code  : #80004E28 - #80004FE7 (  448 )
Bptrl  queue)              Freespace           : #80004FE8 - #800FFFFF ( 1005K)
Fptr0 (high    Empty
Bptr0  queue)              Total memory usage  : 23912 bytes (24K)
Tptrl (timer   Empty
Tptr0  queues) Empty       On-chip memory (4K) : #80000000 - #80000FFF
Clockl (low)   #000C2DD6   MemStart            : #80000070
Clock0 (high)  #030B757C
ParityError    Hard 1011   Debugger has enough memory for 283 processors
ParityAddr     #80005DF0


Last instruction was : in

Option (? for help) (A,C,D,E,F,G,H,I,K,L,M,N,O,P,Q,R,T,V,X,?) ?
```

Figure 9.3 Example post-mortem startup display for a T426 processor

Items displayed on the startup page and their meanings are summarized in table 9.3. Most of the data displayed is common to all transputer types. Where the display differs for specific processor types and debugging modes, this is indicated in the table.

Item displayed	Description
`Iptr`	Instruction pointer (address of the last instruction executed).
`Wdesc`	Process descriptor (process priority and workspace pointer).
`IptrIntSave`†	Saved low priority instruction pointer, if applicable.
`WdescIntSave`†	Saved low priority process descriptor, if applicable.
`A Register`‡	Contents of A register, if known.
`B Register`‡	Contents of B register, if known.
`C Register`‡	Contents of C register, if known.
`Error`	Status of transputer error flag.
`FPU Error`	Status of FPU error flag (T80x series only).
`Halt On Error`	Status of halt on error flag.
`Fptr1`	Front pointer to low priority process queue.
`Bptr1`	Back pointer to low priority process queue.
`Fptr0`	Front pointer to high priority process queue.
`Bptr0`	Back pointer to high priority process queue.
`Tptr1`	Pointer to low priority timer queue.
`Tptr0`	Pointer to high priority timer queue.
`Clock1`	Value of low priority transputer clock.
`Clock0`	Value of high priority transputer clock.
`ParityError`†	Status of parity error register, if applicable.
`ParityAddr`†	Address of parity error, if applicable.
† Not available in breakpoint mode. ‡ Not available in post-mortem mode. Not known to the debugger in breakpoint mode on processors with no hardware support for breakpointing.	

Table 9.3 Data displayed at the monitor page

Process Workspace or Stack

A process workspace (or stack) consists of a vector of words in memory. It is used to hold local variables of the process. The workspace is organized as a falling stack, with 'end of stack' addressing; that is the local variables of a process are addressed as positive offset from the workspace pointer (**Wptr**).

Process Descriptors

In order to identify a process completely, it is necessary to know both its workspace pointer **Wptr** (in which the byte selector is always 0), and its priority (which is 0 for high priority and 1 for low priority). A process descriptor, **Wdesc**, is the sum of the process's workspace pointer, **Wptr**, and its priority.

Process pointers

Iptr points to the last instruction executed and **Wdesc** contains the process descriptor. The saved low priority **Iptr** and **Wdesc** are also displayed if the processor was running a high priority process when it was halted. An asterisk placed next to either an **Iptr** or **Wdesc** indicates an invalid memory location for the process. A double asterisk indicates that the address is outside the defined memory map of the processor. A **Wdesc** value of '**NotProcess**' indicates that no process was executing on the processor when it halted

Practical notes:

- If **Wdesc** contains the value '**MemStart**' it is likely that the **Analyse** signal has been asserted more than once on the network. This can occur on transputer boards where the subsystem signal is asserted on analyze, as on the IMS B004. For further guidance on the use of such boards refer to chapter 4 in the accompanying *Toolset Reference Manual*.
- If **Wdesc** contains the word '**NotProcess**' it means that there were no runnable processes at that instant on the transputer (check timer and external links for any waiting processes) — this may also occur in the presence of deadlock.
- If **WdescIntSave** contains the word '**NotProcess**' it means that a low priority process was not interrupted when the high priority process started running.

Fptr and **Bptr** point to the process run queues, which hold information about processes awaiting execution. The suffix **0** indicates the high priority queue and the suffix **1** indicates the low priority queue.

If the front and back pointers are the same then only one process is waiting; if there are no processes waiting the pointers have no value and the queue is shown as '**Empty**'.

Tptr0 and **Tptr1** are pointers to the high and low priority timer queues respectively.

Registers

In breakpoint mode only, the contents of the transputer registers **Areg**, **Breg**, and **Creg** are displayed for those processors which have built in instructions for breakpoint handling (see table 9.2). Values displayed are those which were current when the process stopped.

Error flags

Two flags are displayed for all processors: **Error** and **HaltOnError**. The **FPError** flag is also displayed for transputers with an integral floating point unit (IMS T80x series).

Clocks

Clock0 and **Clock1** display the values of the high and low priority clocks when the process was stopped. In breakpoint mode the clock values, queue pointers and link information can be updated using the monitor page $\boxed{\text{U}}$ command.

Parity errors

ParityError and **ParityAddr** are only displayed for a T426 processor in post-mortem mode. **ParityError** is the state of the **ParityErrorReg** and can contain one of the following:

Soft *xxxx*	A soft parity error has occurred
Hard *xxxx*	A hard parity error has occurred

The value *xxxx* shows the byte selector bits of the error registers; the value is in binary with byte 3 on the left through to byte 0 on the right. Thus, the value 1011 would show that bytes 3, 1, and 0 are in error.

NotInMem	The memory in a dump file does not include the parity registers
Clear	No parity error has occurred

ParityAddr shows the state of the **ParityErrorAddressReg** and can contain one of the following:

#hhhhhhhh	Word address, in hexadecimal, of location where error occurred
NotInMem	The memory in a dump file does not include the parity registers
Undefined	No parity error has occurred

Memory map

The memory map display is included on the standard startup display — this is the same memory map as displayed by the monitor page $\boxed{\text{M}}$ command. Any or all of the following memory segments may be displayed, depending on the application program and its configuration:

Runtime kernel
Reserved memory
Configuration code
Stack (Workspace)
Program code
Vectorspace
Static area
Heap area
Configuration code
Freespace

When the memory map is displayed, the mode that the debugger is running in is shown. This will be one of:

`Interactive Mode`	When interactively debugging a program.
`Postmortem Mode`	When debugging a program in post-mortem mode.
`Interactive Postmortem`	When post-mortem debugging a program which was previously debugged interactively.
`Dummy Session`	When the debugger is started with the `D` command line option.

9.10.2 Monitor page commands

Most monitor page commands are single-letters that are typed at the monitor page `Option` prompt. A few commands are mapped onto specific function keys. The commands that support breakpoint debugging are only available when the debugger is run in interactive mode.

The main monitor page commands allow you to disassemble and display transputer memory, locate and debug processes, and examine the network processor by processor.

The main commands for common debugging operations are introduced in the following sections. Full details of all the commands can be found in chapter 4 of the accompanying *Toolset Reference Manual.*

Examining memory

Specific segments of transputer memory can be displayed in hexadecimal, ASCII, any high level language type, or disassembled into transputer instructions. The segment of memory to be displayed is specified by a starting address. A map of the transputer's memory can be displayed giving the positions of code and workspace. Commands for examining transputer memory are summarized below.

$\boxed{\text{A}}$	Display memory in ASCII.
$\boxed{\text{D}}$	Disassemble into transputer instructions.
$\boxed{\text{H}}$	Display memory in hexadecimal.
$\boxed{\text{I}}$	Display memory in selected data type.
$\boxed{\text{M}}$	Memory map.

Locating processes

Locating to code for specific processes is one of the major functions available through the monitor page. The commands available allow processes other than the stopped or current process to be located and examined anywhere on the network. Processes can be located on the current processor by examining run queues, and on other processors by jumping down transputer links.

Four commands are used, three to display waiting processes and one to jump to the selected code of a process displayed by the other three.

R	Display processes waiting on Run queues.
T	Display processes waiting on Timer queues.
L	Display processes waiting on transputer Links.
Z	Display processes waiting on software virtual links.
G	Goto symbolic debugging for the selected process.

These commands can be used to trace all processes on a network and determine the cause of program failure. The method is explained in more detail in section 9.11.

Specifying processes

The O command allows a specific process to be selected for symbolic debugging, providing the address is known.

O	Specify a process for symbolic debugging.

This command is useful for switching directly to symbolic debugging for a process whose instruction pointer and process descriptor you have already noted, earlier in the debug session.

Selecting processes

The F command enables a source file to be selected for symbolic display using the filename of the *object* module produced for it.

F	Select a source file to be displayed.

This option enables symbolic locating (for setting breakpoints etc.) without needing to know `Iptr` and `Wdesc` process details (as the G and O commands do).

Other processors

Two commands and two cursor keys allow other processors to be selected.

E	Go to next halted processor.
P	Go to specified processor.

◀	Go to the next lowest numbered processor.
▶	Go to the next highest numbered processor.

The sequence of processors used by the E and cursor key commands is an internal sequence read by the debugger. Processor numbers corresponding to visible names in the configuration file can be determined by using the K command.

Breakpoint commands

The following commands support breakpointing. To use these commands the debugger must be run with the 'B' command line option.

B	Breakpoint menu.
J or RESUME	Jump into and run application program.
S	Show debugging messages and prompts menu.
U	Update processor status display.
W	Write value to memory.

Changing to post-mortem debugging

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

Y	Postmortem debug current breakpoint session.

9.11 Locating processes

Most transputer programs consist of several processes running in parallel, either on the same transputer or on separate processors connected by their INMOS links.

If a program error halts the transputer then the debugger automatically locates to the stopped process, which can then be examined directly. If the program runs incorrectly but does not halt the processor, a good approach is to locate to and examine each process in turn.

There may be many processes running on the transputer when it is interrupted from the keyboard, or the `idump` tool is run to create a dump file for debugging. Each process exists in one of a number of possible states:

- Not yet started.
- Running on the processor.
- Waiting on a process execution queue (Run queue).
- Waiting on a timer queue.
- Waiting for communication from another process on the same processor.
- Waiting for communication on a transputer link (Link information).
- Interrupted by a high priority process.
- Already stopped or terminated.

9.11.1 Running on the processor

One, and only one, process may execute on the transputer at any instant. The debugger will automatically locate to this process (if there was one) when the debugger is executed. All other processes are either waiting, stopped, or not yet started.

9.11.2 Waiting on a run queue

Processes on the run queues (i.e. waiting to be executed) can be located by first using the monitor page $\boxed{\text{R}}$ command to display the list of waiting processes. A process can be selected from the list by pressing $\boxed{\text{G}}$ (for 'Goto process'), moving the cursor to the appropriate address, and then pressing $\boxed{\text{RETURN}}$. Processes can also be located to by specifying the displayed `Iptr` and `Wdesc` with the $\boxed{\text{O}}$ command.

The values displayed with the $\boxed{\text{R}}$ command can be used to determine the overall status of run queues. If no processes are waiting then the content of the queue is shown as '`Empty`'. If pointer addresses are displayed then there are processes waiting; if the front and back pointers have the same value then there is only one process waiting.

9.11.3 Waiting on a timer queue

Processes waiting for a specified time are placed on the high and low priority timer queues. These are similar to the run queues except that they are controlled by the transputer clocks.

In a similar way to processes on the Run queues, processes on the timer queues can be located by using the monitor page $\boxed{\text{T}}$ command to display a list of processes and then using the $\boxed{\text{G}}$ command, or by specifying the process address. Pointers to the timer queues indicate overall queue status in a similar way to the run queues.

9.11.4 Waiting for communication on a link

Processes waiting for a hardware communication (input or output on a transputer link, or an input on the **Event** pin) can be located by using the monitor page $\boxed{\text{L}}$ command to display a list of waiting processes, and then using the $\boxed{\text{G}}$ command to locate to the process. Links where no processes are waiting are shown as '**`Empty`**'.

At most 9 processes can be waiting for a hardware communication, two for each of the four links and one on the **Event** pin.

See section 9.4.1 for information on the restrictions on locating down hard channels.

9.11.5 Waiting for communication on a software virtual link

Processes waiting for a communication on a software virtual link (as provided by the configurer) can be located by using the monitor page $\boxed{\text{Z}}$ command to display a list of waiting processes, and then using the $\boxed{\text{G}}$ command to locate to the process. Virtual links where no processes are waiting are shown as '**`Empty`**'.

This is the preferred method for locating processes waiting on external communications when software virtual links are present.

9.11.6 Waiting for communication on a channel

Processes waiting for a communication on a channel can be located from source level using the $\boxed{\text{CHANNEL}}$ function. This function works for both internal (or *soft*) channels and external (or *hard*) channels (channels mapped onto processor links).

Only one process can be waiting on a channel. If no process is waiting, the channel is shown as '**`Empty`**'.

9.11.7 Interrupted by a high priority process

A low priority process may have been interrupted by a high priority process. Such a process may be selected using the $\boxed{\text{G}}$ or $\boxed{\text{O}}$ commands and the values stored in the **`WdescIntSave`** location.

9.11.8 Processes terminated or not started

Processes which have stopped executing, or not yet started, do not have process descriptors and so they cannot be examined by the debugger. If the currently running process and all the waiting processes have been found (not forgetting all those processes waiting on all the internal channels) then any processes still unaccounted for must either have already finished or failed to start.

9.11.9 Locating to procedures and functions

When a procedure is called, the workspace pointer is moved. If the debugger locates inside any code of defined scope (such as a procedure) then only local variables, and variables declared globally, are in scope and available for inspection.

To inspect variables or channels not in scope within the procedure or function, use $\boxed{\text{BACKTRACE}}$ to locate to a position where the desired variable or channel is in scope. To relocate back into the procedure or function use $\boxed{\text{RETRACE}}$ to undo each backtrace, or $\boxed{\text{TOP}}$ to return to the initial location.

9.12 Debugging support library

Three routines are provided in the libraries to assist with debugging. These provide the functions *stop*, *assert*, and *message*. The routines have different names for each language and are described in more detail in the appropriate *Language and Libraries Reference Manual*. Table 9.4 summarizes the routines for each language.

Routine		Description
`debug_assert`	C	If the parameter evaluates to false then stop the process and inform the debugger.
`DEBUG.ASSERT`	occam	
`DEBUG_ASSERT`	FORTRAN	
`debug_stop`	C	Stop the process and inform the debugger.
`DEBUG.STOP`	occam	
`DEBUG_STOP`	FORTRAN	
`debug_message`	C	Insert a debugging message in the program.
`DEBUG.MESSAGE`	occam	
`DEBUG_MESSAGE`	FORTRAN	

Table 9.4 Debug support functions

The stop and assert routines are used to stop a process, the latter on the failure to meet a specified condition; such events are treated as a program error by the debugger. The message is used to insert messages that will only be displayed when the program is run under the interactive debugger.

For C and FORTRAN the procedures are included in the standard library that is incorporated at link time and are directly accessible from the program without further action by the programmer. For occam programs, the library `debug.lib` must be referenced with a `#USE` in the source code and also included as an input to the linker.

In the following descriptions the C versions of the functions are used; the descriptions apply equally to the respective occam and FORTRAN versions.

`debug_assert()` and `debug_stop()` allow a process to be stopped at any point in the code, where it can then be debugged using the symbolic functions and Monitor page commands. `debug_stop()` always stops the process whereas `debug_assert()` only stops the process if the parameter evaluates to false.

`debug_message()` is used to insert debugging messages into the code. Messages are relayed back to the terminal from any point in the program, even from code running on distant processors of a network. It can be used to monitor the activity of outlying processors which are not directly connected to the host. The display of debug messages at the terminal is controlled by an option on the Monitor page Breakpoint Menu (the default is to display them). **Note**: Only the first 80 characters of a message are displayed.

9.12.1 Examples

The use of the debug support functions is illustrated in the C and occam examples below. Sources may be found in `examples/manuals/idebug`.

C example:

```
/*****************************************
*
*  Debugger example:  debug.c
*
*  Example of debug support functions when used with
*  and without the debugger.
*  (see also debugger example file abort.c)
*
*****************************************/

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

int
main (void)

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

      printf ("Program started\n");

      debug_message ("A debug message only within the debugger");

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

      printf ("Program being halted by debug_stop ()\n");
      debug_stop ();

      exit (EXIT_SUCCESS);
}
```

In this example, if **x** is 1 **debug_assert** evaluates to true and the program runs until it encounters **debug_stop**. If **x** is 0 (as in the example) **debug_assert** evaluates to false and the process stops before it reaches **debug_stop**. Code stopped by **debug_assert** and **debug_stop** may be resumed from the line following the call of the debug function using the CONTINUE FROM key.

occam example:

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

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

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

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

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

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

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

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

In this example `x` is set to FALSE, therefore `DEBUG.ASSERT` evaluates to FALSE and the process stops before it reaches `DEBUG.STOP`. If `x` were set to TRUE `DEBUG.ASSERT` would evaluate to TRUE and the program would run until it encountered `DEBUG.STOP`. Code stopped by `DEBUG.ASSERT` or `DEBUG.STOP` may be resumed from the line following the call of the debug function using the CONTINUE FROM key.

9.12.2 Actions when the debugger is not available

If the debugger is not available on the system the debug library procedures have the following actions:

`debug_assert`
`DEBUG.ASSERT`
`DEBUG_ASSERT`
	If the parameter evaluates to false then stop the process (also stops the processor if configured in HALT mode).

`debug_stop`
`DEBUG.STOP`
`DEBUG_STOP`
	Stop the process (also stops the processor if configured in HALT mode).

`debug_message`
`DEBUG.MESSAGE`
`DEBUG_MESSAGE`
	No action.

9.13 Debugging with `isim`

The T425 simulator `isim` provides a single processor interactive simulation of a program running on an IMS T425 transputer, on a boot from link transputer board connected to a host computer through the host file server `iserver`. The interactive environment provides a machine level (non-symbolic) environment, similar to the debugger monitor page, for debugging programs and monitoring program execution.

The simulator allows any single processor program to be run and analyzed without a transputer board. All the component parts of a program to be simulated, must be compiled for the T425 transputer type (or compatible class — see appendix B of the accompanying *Toolset Reference Manual*).

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

9.13.1 Command interface

The simulator has a single command interface which corresponds to the debugger monitor page. Most commands are single letter commands and can be executed with a single key press. For a list of commands see chapter 14 in the accompanying *Toolset Reference Manual.*

9.13.2 Using the simulator

The simulator can be used in two ways:

- To debug programs by inspection of the transputer and memory, in the same way as with the debugger. Registers, memory, and machine state can be examined directly at the monitor page.
- To monitor the execution of programs using machine level single step execution and the setting of break points at specific memory locations. Code can be executed by stepping single transputer instructions.

9.13.3 Program execution monitoring

The simulator provides a number of functions that can be used interactively to monitor and control the behavior of a program. These are:

- Breakpoints
- Single step execution of a program

Breakpoints

Breakpoints can be set, displayed, and cancelled using the '`B`' command to display the Breakpoint Options Page.

Single step execution

A program can be stepped a single transputer instruction at a time using the 'S' command.

9.13.4 Core dump file

`isim` may be used to produce a core dump file that can be read by the debugger (as if the code had been executed on a real transputer and the memory dumped using the `idump` tool).

9.14 Hints and further guidance

This section gives some further guidance on some specific points related to use of the debugger.

9.14.1 Invalid pointers

The debugger checks process instruction pointers (`Iptr`) and process descriptors (`Wdesc`) for the correct code and data limits. Invalid pointers are flagged by an asterisk (*) on the screen. Invalid pointers outside the processor's memory are flagged with a double asterisk ('**').

Invalid pointers can indicate a major problem with the program. They may also be caused by specifying an incorrect dump file.

9.14.2 Examining and disassembling memory

Within the monitor page environment, the debugger keeps a record of two memory addresses; the start address of the last disassembly, used as the default by the $\boxed{\text{D}}$ command, and the address of the last region of memory to be displayed, used by the $\boxed{\text{A}}$, $\boxed{\text{H}}$ and $\boxed{\text{I}}$ commands.

This allows you to switch easily between code disassembly and memory display. You can, for example, disassemble a portion of memory using the $\boxed{\text{D}}$ command, examine its workspace in hex using the $\boxed{\text{H}}$ command, and then return to the original address by using the $\boxed{\text{D}}$ command once again.

9.14.3 Scope rules

The debugger can only display variables that are in scope at its current location point in the source code.

9.14.4 Inspecting soft configuration channels

Soft channels declared at the configuration level (i.e. those internal to a processor which are not placed on its external links) may be inspected from the monitor page by knowing that they are located near the beginning of the *Configuration code* area which appears after the user *Program code* area (as displayed by the monitor page Memory map command).

9.14.5 Locating to IF, ALT and CASE in occam

`IF` and `ALT` 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 cases like these there is no obvious statement to locate to and the debugger locates instead to the *start* of the construct.

When using these constructs it is good practice to always define the default case. The debugger can then locate directly to the `STOP` statement where the error occurred.

9.14.6 Analyzing 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. This is illustrated below using occam; programs could be similarly written in C or FORTRAN.

Sources may be found in `examples/manuals/idebug`.

Consider the following code which creates a deadlock:

```
------------------------------
--
--  Debugger example:  deadlock.occ
--
--  Example of deadlock.
--
------------------------------


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


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

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

      INT x :
      SEQ
        c ? x
  :                  --  <==  Missing second input

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

The program can be debugged by adding a process that will remain idle (here, waiting on a `TIMER`) while the program is debugged. An example of the type of code that is required is illustrated below.

```
------------------------------
--
--  Debugger example:  deadfix.occ
--
--  Example of deadlock and how to provide
--  debugging support.
--
------------------------------


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


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

  PROC deadlock.debug ()
    CHAN OF INT c :
    CHAN OF INT stopper :
    PAR
      DEBUG.TIMER (stopper)    --  Hook for debugger
      SEQ
        PAR
          SEQ
            c ! 99
            c ! 101

          INT x :
          SEQ
            c ? x
            --  <==  Missing second input

        stopper ! 0  --  terminate DEBUG.TIMER
  :

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

The procedure `DEBUG.TIMER` is supplied in the occam debugging library. Similar routines could be written for other languages, and the principle of operation is the same – the process lies dormant on the processor's timer queue waiting for a time as far into the future as the processor can provide. When the timeout expires, the process places itself back on the timer queue. Such a process provides a hook into the program for locating deadlocked processes because the process is always accessible to the debugger on the timer queue. By locating to it you can access variables which are in scope at the point of its execution and thereby detect the deadlock. In the modified program a deadlock still forms in the procedure, but there is now a way to enter the program.

To enter the program and inspect the deadlock, first invoke the Monitor page environment, and use the Monitor page 'T' command to inspect the transputer's timer queue, on which there will be a process waiting. Use the 'G' command to go to that waiting process, and the debugger will locate to the call of `DEBUG.TIMER`.

You can then use $\boxed{\text{INSPECT}}$ 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 $\boxed{\text{CHANNEL}}$ 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 that the runtime code should not be needlessly altered. 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 always be true. Lastly, it could be difficult to distinguish the true deadlocked process from the many idle debug processes waiting on the timer queues.

9.15 Points to note when using the debugger

This section contains some extra information which may be of use when using the debugger. Sections 9.15.1 to 9.15.18 apply to debugging in all languages; section 9.15.19 gathers together those aspects which apply only to C.

9.15.1 Abusing hard links

Current generation transputers permit unsynchronized transfer of messages on external channels (links). This allows, for example, two 4-byte messages to be sent and for them to be received as a single 8-byte message on the receiving transputer. This is not consistent with the communication of messages between processes on the same processor where the transfer of messages is synchronized.

When breakpoint debugging, external communications are handled by the debugger's virtual link system; this involves an internal transfer which will function incorrectly if user code is relying on unsynchronized transfers. Unsynchronized data transfer should not be used where breakpointing is used to debug a program. It is bad practice anyway and will certainly cause the virtual link system (used by both the debugger and the virtual-routing configurer) to crash.

9.15.2 Examining an active network (the network is volatile)

When a process stops at a breakpoint you should remember that all of the other processes are still running (unless they hit a breakpoint, terminate etc.). This means that data displayed by any of the monitor page commands that display process queues, etc. (e.g. $\boxed{\text{R}}$, $\boxed{\text{L}}$, $\boxed{\text{T}}$ etc.) may change if they are re-displayed (e.g. by using the same command again or the $\boxed{\text{U}}$, Update, command to update the displayed information).

When in symbolic mode the same is true for channels which may appear empty when first inspected only to change to a waiting process when inspected again. The only way to effectively *freeze* all processes is to flip to post-mortem mode by using the monitor page $\boxed{\text{Y}}$ (Enter Postmortem Mode) command. You should remember that when you use this command that all processes that have hit a breakpoint will not appear in the runtime queues. If this is a problem, you should note the `Iptr` and `Wdesc` values of the processes and, when in post-mortem mode, use the monitor page $\boxed{\text{O}}$ (Select Process) command to locate to them symbolically.

9.15.3 Using $\boxed{\text{INSPECT}}$ with channel communications

When debugging a program compiled for interactive debugging it should be remembered that channel communication is achieved via library calls. As a consequence, the $\boxed{\text{INSPECT}}$ key may display an `Iptr` relating to code in the debugging kernel system rather than the `Iptr` of a user process waiting on the channel. This may lead to several channel communications appearing to having the same process `Iptr` (the `Wdesc` will be valid and unique). In order to correctly establish the `Iptr` of the process waiting at the other end, you should use the $\boxed{\text{CHANNEL}}$ key to locate to the process followed by the $\boxed{\text{INFO}}$ key to obtain process details.

9.15.4 Debugging in the presence of software virtual links

When the configurer creates software virtual links it places additional processes onto the processor in order to provide the virtual link services. These processes will be displayed by the debugger — it displays all processes it finds on the run queue, links etc. A consequence of this is that, occasionally, a process will be displayed which forms part of the software virtual link system. It is not possible locate to these processes (as they are is not part of the program being debugged). These processes may be identified by noting the `Iptr` and `Wdesc` values and using the $\boxed{\text{V}}$ command to search for a process with a code area which contains the `Iptr` value, and a stack area which contains the `Wdesc` value. If the name of the process is "`%ROUTER[]`" then it is a software virtual link process which you may not locate to.

A similar problem occurs when attempting to locate to a process waiting on a transputer link which is used by the software virtual link system — the debugger will complain that it cannot find a file with a name such as "`vrdebxx.tco`" (where *xx* is a sequence of digits).

Another problem encountered with using software virtual links and `idebug` is that low priority user processes are promoted, temporarily, to high priority when they communicate on software virtual links The debugger cannot tell if they were originally at high or at low priority: it will locate to what it believes is a high priority process. In general, this is not a problem if you wish to inspect variables etc. If this does present a problem and you know that a particular process is a low priority process, you should use the $\boxed{\text{O}}$ command and specify a low priority `Wdesc` when prompted, by setting the least significant bit of the `Wdesc` value of the process (e.g. `%1234` becomes `%1235`).

In general, the preferred method for locating processes waiting on external communications when software virtual links are present is the Monitor page $\boxed{\text{Z}}$ command. If however, you know that a transputer link is not used for software virtual routing, you should use the Monitor page $\boxed{\text{L}}$ command to locate to such processes.

9.15.5 Selecting events from specific processors

The debugger provides no guarantee that debugging events, such as breakpoints and debugging messages, from processes running on different processors are presented in the same that order they occur in. Events on processors which are closer, in terms of connectivity, to the root transputer (where the debugger is running) are usually displayed before events on more distant processors.

If it is important that you encounter a debugging event on a specific processor before events on other processors, you can usually achieve this by changing to the processor of interest (using the monitor page $\boxed{\text{P}}$ command or left and right cursor keys) *before* resuming via the $\boxed{\text{J}}$ or $\boxed{\text{RESUME}}$ command.

9.15.6 Minimal confidence check

A first level confidence check to perform with a program which is misbehaving is to perform a 'compare memory' check using the monitor page $\boxed{\text{C}}$ command. This will help to highlight any memory corruption problems which may occur due to faulty memory or faulty program logic. If using occam, you can prevent out of range accesses to memory by ensuring that no compiler checks have been disabled.

9.15.7 INTERRUPT key

The debugger can be diverted from the running program to return to the monitor page by the use of the $\boxed{\text{INTERRUPT}}$ key. However, problems can arise if the running program is simultaneously trying to read from the keyboard; the debugger is then unable to intercept the interrupt key. (Sometimes it is possible to force the interrupt to be recognized by repeating the key quickly.)

A similar problem arises when there are existing keystrokes buffered before the interrupt key; if the application program does not read these buffered keystrokes the debugger will never have a chance to see the interrupt key.

Note: The INTERRUPT key will disable all `iserver` requests to the application until the debugger is directed to resume the application.

9.15.8 Program crashes

If the debugger detects that the program has crashed immediately after starting program execution (i.e. after the J, Jump into application, command), you should use the post-mortem debug command, Y, to determine the cause. However, if no error flags are set on the network that is running the program then it is likely that an error flag is set on a transputer that is not in use. This may occur on boards where the subsystem services are wired to propagate all error flags to the root transputer. In this instance you need to clear all the error flags in the network (see section 9.7.4).

9.15.9 Undetected program crashes

When operating in breakpoint mode and a program overwrites the debugging kernel or you have set a breakpoint in a high priority process on a processor without hardware breakpoint support, the debugger cannot fully recover and is unable to indicate that the program has crashed. In this situation the debugger fails to update the screen other than to put the following message at the top of the screen when it attempts to display the monitor Page:

```
Toolset Debugger : V4.00.00 Processor n "name" (Txxx)
```

In such instances you should use the host BREAK key in order to terminate the debugger and restart the debugger using the command line '`M`' option to post-mortem debug the session.

9.15.10 Debugger hangs when starting program

If the debugger hangs immediately after you have supplied the command line arguments when starting execution of a program you have probably set a breakpoint in a configuration-level, high priority process on a processor without hardware breakpoint support.

9.15.11 Debugger hangs

If the debugger hangs when attempting to flip to post-mortem mode using the monitor page Y command, or when trying to quit, you should terminate the debugger manually using the host BREAK key. If you were trying to switch to post-mortem mode you should restart the debugger using the command line 'M' option to resume debugging in post-mortem mode.

9.15.12 Catching concurrent processes with breakpoints

Sometimes a concurrent process is executing in a program (often in a loop) and you would like to be able to control it better by using breakpoints. If the process is communicating with other processes via channels, and you have set breakpoints in these other processes, then breakpoints can be set on a communication and, when you hit that breakpoint, the channel can be jumped down to debug the executing process.

However, if the process has entered a non-communicating loop or you are not sure where exactly it is in your program code, you must use a different approach. In order to set a breakpoint, you should use the INTERRUPT key to return to the monitor page and then, by using the R (Run queues) command and/or the T (Timer queues) command, list the `Iptrs` and `Wdescs` of the processes currently executing. (Often, this will include the debugging kernel processes but these are easy to detect because they are marked as kernel processes.)

Use the G (Goto process) command to select the `Iptr` and `Wdesc` to locate symbolically to the process. You can then set a breakpoint on that line, return to the monitor page and resume the program using the J or RESUME command; when the process hits the breakpoint you may continue to debug it. If there are no processes on either the run or timer queues and there are no external communications, it means that your program has either deadlocked or terminated.

9.15.13 Phantom breakpoints

Because of the mechanism used for breakpoints on those transputers without hardware breakpoint support (see table 9.2) it is possible for the output from the INMOS compilers to contain code that fools the debugger into thinking it is a breakpoint (a *phantom* breakpoint). This happens when the code contains an empty loop that does not terminate. The following code examples will generate phantom breakpoints:

C	occam	FORTRAN

`while (1){`
`      ;`
`}`
	`WHILE TRUE`
	`  SKIP`
		`DO WHILE TRUE`
		`END DO`

`for (;;){`
`      ;`
`}`
		`100   GOTO 100`

If you encounter a phantom breakpoint and you wish to continue execution, you must set a breakpoint at the same address and then resume execution. To do this use the GET ADDRESS key to obtain the start address of the empty loop when in symbolic mode, change to the monitor page and use the Set Breakpoint option on the Breakpoint menu to set a breakpoint at the loop address.

9.15.14 Breakpoint configuration considerations

When breakpoint debugging you should remember that the root transputer of a network is used by the debugger for its own purposes. On some transputer motherboards with an built-in pipeline, the root transputer is normally booted down link 0; subsequent transputers in the pipeline boot down link 1. This may (accidentally) be a problem if you simply take a network configuration which was not configured with breakpoint debugging in mind (e.g. a pipeline configuration) and attempt to breakpoint debug it. The debugger will in effect, attempt to skip load it onto the rest of the network; the program may load (if by chance the right link connections are available) but, if the boot link is different, it will not be able to talk to the host (via **iserver**) when it executes.

Such a problem may easily be checked for by using the monitor page L command when positioned on processor 0. This will indicate whether the root transputer was booted from a different link to that specified in the configuration file.

When breakpoint debugging, the debugger will warn you if the boot link is different from that expected for the root processor before the network is loaded.

9.15.15 Determining connectivity and memory sizes

In order to establish the connectivity and memory map range for each processor in a program you should use the **icollect** 'P' option. Alternatively you may use the debugger command line option 'D' (dummy debug).

9.15.16 Long source code lines

Source code lines longer than 500 characters cause the symbolic source code browser to treat them as multiple lines and subsequently it will loose line synchronization; (i.e. it displays incorrect line number information).

9.15.17 Resuming breakpoints on the transputer *seterr* instruction

If an attempt is made to resume from a breakpoint which is at the address of a *seterr* instruction, the debugger does not continue with the original (correct) `Iptr` (it resumes with an `Iptr` within the kernel area). Because the debugger operates in Halt-on-Error mode, the *seterr* instruction will halt the processor.

The effect of the incorrect `Iptr` is only apparent if you subsequently switch to post-mortem debugging whereupon the debugger will complain that it is unable to locate to an `Iptr` within the kernel area. If this is a problem, you should note the `Iptr` before resuming from the breakpoint.

Setting and resuming breakpoints on an occam **STOP** statement compiled in HALT mode, will cause this problem.

9.15.18 Shifting by large or negative values

The shift instructions on current transputers take time proportional to the number of places shifted — as this number is unsigned, negative values will be treated as large positive values. Large shifts will cause current transputers to temporarily 'lock' for a number of cycles equal to the number of places shifted — on 32 bit transputers this can cause the device to hang-up for up to 2^{32} cycles (approximately $3^1/_2$ minutes for a 20 MHz device).

Some languages, such as C, performs no runtime checks for invalid shift values and so do not protect you against their consequences. Other languages, such as occam, do perform such checks.

If the debugger, in post-mortem mode, locates to a source line containing a shift operator and the error flag has not been set then it is likely that a shift by a large value is taking place — this can be verified by using the INSPECT key to check the shift count.

9.15.19 Aspects of C debugging

Arrays as arguments to C functions

Because C requires a declaration of a parameter as *array of type* to be adjusted to *pointer to type* the debugger must treat all array parameters as pointers. This means that it cannot automatically display the contents of an array passed as a parameter.

In order to display the contents of arrays you should use specify the range of the array to be displayed. This is illustrated in the following example.

```
void foo (int p[4]) {
        debug_stop ();
}
```

The argument `p` will be treated as a pointer to `int` rather than an array of `int` by the C compiler. Using the INSPECT function on `p` will cause the *address* of `p` to be displayed. In order to see the *contents* of the array, the inspect command should be given an array range, for example: `p[0;3]`.

Backtracing with concurrent C processes

`idebug` supports backtracing from a parallel process to the parent process (where the parallel process was started via a C library call). However, for processes started asynchronously via `ProcRun`, `ProcRunHigh`, or `ProcRunLow`, `idebug` merely enables you to backtrace and does not allow operations such as inspection of variables after a backtrace. This is because the parent process which started the asynchronous processes may no longer exist, in which case inspection is meaningless.

Errors generated by the full C library

Generally, the full C runtime library is able to detect when there is insufficient memory for it to function correctly; in such instances it displays an error message at startup.

In rare circumstances the library is able to detect that there is insufficient memory but it does not have enough memory to display the startup error message. In such instances, it sets the error flag and terminates execution. If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger then it is highly likely that insufficient memory is available for either the static or the heap area:

```
Error: Not compiled with debugging enabled "libc.lib"
```

Errors generated by the reduced C library

Because the reduced C runtime library has no host to communicate with, if a runtime error occurs the reason for the error is not readily apparent. If a program sets the error flag and the debugger is unable to backtrace when the last instruction executed was *seterr* (error explicitly set), and the following error message is displayed by the debugger then it is highly likely that insufficient memory is available for either the static or the heap area:

```
Error: Not compiled with debugging enabled "libcred.lib"
```

C compiler optimizations

The INMOS compilers perform some code optimizations. If an external variable is optimized out from a module because it is never used then the debugger is informed of this and is able to relay the information to the user.

However, for some optimizations the debugger is not informed and consequently it may provide misleading information. The following code illustrates this:

```
int main (void){

        int     a = 0;
        int     b = 0;

        while (1) { /*  or  'for (;;)'  */
             ;
        }

        /* following code optimized out by compiler
         * as it can never be reached
         */
        a = 42;
        b = a + 1;
        a = b * b
        ...
}
```

In these cases the debugger may show the discrepancy in either of the following ways:

1 If a function follows the optimized code, the debugger associates the address of the optimized lines with the address of the start of the function.

2 If no function follows the optimized code then the debugger indicates that it is unable to find the address for any of the optimized lines.

9.16 C debugging example

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

A similar example program written in occam is described in section 9.17.

9.16.1 The example program

The example program `facs.c` calculates the sum of the squares of the first *n* factorials, using a rather inefficient algorithm. It has been structured this way for clarity in process structure and to demonstrate parallel processing and debugging methods. The same program coded in occam is supplied with the occam 2 toolset. The program incorporates five processes, each coded as a separate function. The five processes in turn input *n*, calculate factorials, square the factorials, sum the squares, and output the result. The program is listed below.

```
/*****************************************
*
*  Debugger example:  facs.c
*
*  idebug (and parallel C) example based on similar program
*  in occam toolset.
*
*  Uses 5 processes to compute the sum of the squares of the
*  first N factorials using a rather inefficient algorithm.
*
*  Plumbing:
*
*  - > feed -> facs -> square -> sum -> control <--> User I/O
*  |                                     |
*   --------------------------------------
*
*****************************************/


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


const double    stop_real    = -1.0;
const int       stop_integer = -1;

/*  output a double down a channel   */
void
ChanOutDouble (Channel *out, double value)

{
        ChanOut (out, (void *) &value, sizeof (value));
}

/*  input a double from a channel  */
double
ChanInDouble (Channel *in)

{
        double  value;

        ChanIn (in, (void *) &value, sizeof (value));
        return value;
}
```

```
/*  compute factorial  */
double
factorial (int n)

{
        double  result;
        int     i;

        result = 1.0;
        for (i = 1; i <= n; ++i)   {
                result = result * i;
        }
        return result;
}


/*  source stream of ints  */
void
feed (Process *p, Channel *in, Channel *out)

{
        int     n, i;

        (void) p;       /*  stop compiler usage warning  */

        n = ChanInInt (in);
        for (i = 0; i < n; ++i)   {
                ChanOutInt (out, i);
        }
        ChanOutInt (out, stop_integer);
}


/*  generate stream of factorials  */
void
facs (Process *p, Channel *in, Channel *out)

{
        int     x;
        double  fac;

        (void) p;       /*  stop compiler usage warning  */

        x = ChanInInt (in);
        while (x != stop_integer)   {
                fac = factorial (x);
                ChanOutDouble (out, fac);
                x = ChanInInt (in);
        }
        ChanOutDouble (out, stop_real);
}
```

```
/*  generate stream of squares  */
void
square (Process *p, Channel *in, Channel *out)

{
        double  x, sq;

        (void) p;       /*  stop compiler usage warning  */

        x = ChanInDouble (in);
        while (x != stop_real)   {
                sq = x * x;
                ChanOutDouble (out, sq);
                x = ChanInDouble (in);
        }
        ChanOutDouble (out, stop_real);
}

/*  sum input  */
void
sum (Process *p, Channel *in, Channel *out)

{
        double  total, x;

        (void) p;       /*  stop compiler usage warning  */

        total = 0.0;
        x = ChanInDouble (in);
        while (x != stop_real)   {
                total = total + x;
                x = ChanInDouble (in);
        }
        ChanOutDouble (out, total);
}
```

```
/*  user interface and control  */
void
control (Process *p, Channel *in, Channel *out)

{
        double  value;
        int     n;

        (void) p;       /*  stop compiler usage warning  */

        printf ("Sum of the first n squares of factorials\n")
;
        do   {
                printf ("Please type n : ");
        }   while (scanf ("%d", &n) != 1);
        printf ("n = %d\n", n);
        printf ("Calculating factorials ... ");

        ChanOutInt (out, n);
        value = ChanInDouble (in);

        printf ("\nThe result was : %g\n", value);
}


Channel *
Checked_ChanAlloc ()

{
        Channel *chan;

        if ((chan = ChanAlloc ()) == NULL)   {
                fprintf (stderr, "ChanAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return chan;
}


Process *
Checked_ProcAlloc (void (*func)(), int sp, int nparam,
                        Channel *c1, Channel *c2)

{
        Process *proc;

        proc = ProcAlloc (func, sp, nparam, c1, c2);
        if (proc == NULL)   {
                fprintf (stderr, "ProcAlloc () failed\n");
                exit (EXIT_FAILURE);
        }
        return proc;
}
```

```
int
main (void)

{
        Channel *facs_to_square, *square_to_sum;
        Channel *sum_to_control, *feed_to_facs;
        Channel *control_to_feed;

        Process *p_feed, *p_facs, *p_square;
        Process *p_sum, *p_control;

        facs_to_square  = Checked_ChanAlloc ();
        square_to_sum   = Checked_ChanAlloc ();
        sum_to_control  = Checked_ChanAlloc ();
        feed_to_facs    = Checked_ChanAlloc ();
        control_to_feed = Checked_ChanAlloc ();

        p_feed = Checked_ProcAlloc (feed, 0, 2,
                        control_to_feed, feed_to_facs);
        p_facs = Checked_ProcAlloc (facs, 0, 2,
                        feed_to_facs, facs_to_square);
        p_square = Checked_ProcAlloc (square, 0, 2,
                        facs_to_square, square_to_sum);
        p_sum = Checked_ProcAlloc (sum, 0, 2,
                        square_to_sum, sum_to_control);
        p_control = Checked_ProcAlloc (control, 0, 2,
                        sum_to_control, control_to_feed);

        ProcPar (p_feed, p_facs, p_square, p_sum,
                                        p_control, NULL);

        exit (EXIT_SUCCESS);
}
```

9.16.2 Compiling and loading the example

The source of the program is provided in the toolset `debugger` examples subdirectory. It should be compiled for transputer class TA with debugging enabled, then linked with the appropriate library files and made bootable using `icollect` with the '`T`' option to create single transputer bootable code.

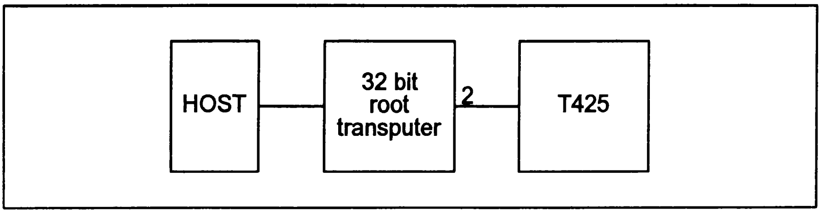


Figure 9.4 Hardware configuration for the example

The example is intended for running on a B008 board wired *subs*. See section 4.7 in the accompanying *Toolset Reference Manual* debugger chapter if your system is different.

A typical sequence of commands for compiling, linking, and booting the program is shown below. The '`I`' option on the linker command line is optional but provides useful information on the progress of the linking operation.

Command sequences are shown for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

UNIX:

```
icc facs.c -g -ta -o facs.tax
ilink facs.tax -f cnonconf.lnk -ta -o facs.cah -i
icollect facs.cah -t
```

MS-DOS/VMS:

```
icc facs.c /g /ta /o facs.tax
ilink facs.tax /f cnonconf.lnk /ta /o facs.cah /i
icollect facs.cah /t
```

The program is loaded for breakpoint debugging by running `idebug` with in interactive mode using one of the following commands:

```
idebug -sr -si -b2 facs.btl -c t425

idebug /sr /si /b2 facs.btl /c t425
```

This command starts up the debugger and displays the Monitor page but does not start the program. The `iserver` '`SI`' switch is optional.

Note: If your transputer is not a T425 you should change the `t425` option to the appropriate transputer type. You may also need to change the number specified after the '`B`' option to the number of the root transputer link to which the network is connected. See table 4.4 in chapter 4 of the accompanying *Toolset Reference Manual* for more details about the options to use, if in doubt.

9.16.3 Setting initial breakpoints

Initial breakpoints can often be set by using the Monitor page $\boxed{\text{B}}$ command and specifying a breakpoint at the start of `main()`. In this example we use a different method based on setting specific breakpoints in the source code before the program is started.

At the Monitor page select $\boxed{\text{F}}$ to display the source file. At the object module filename prompt specify the compiled object file `facs.tax`. The debugger uses debug information within the object module to select the source file.

The source file is displayed with the cursor positioned at the first function definition. At this point the program is still waiting to be started.

Set a breakpoint at the beginning of the `ChanOutDouble()` function using TOGGLE BREAK. The debugger confirms the breakpoint is set and gives the breakpoint a unique identification number (note that the breakpoint is set on the first executable line of the function).

9.16.4 Starting the program

Return to the Monitor page using the MONITOR key and start the program by selecting the J command. Press RETURN at the '`Command line`' prompt (no command line is required) and give a small positive number (e.g. 12) when the program prompts for input. The program runs until it reaches the breakpoint.

9.16.5 Entering the debugger

At the breakpoint the debugger displays the number of the breakpoint and the number of times it has been encountered (or hit) and then requests confirmation to continue the stopped process. Press any key except R or r to enter the symbolic debugging environment. The debugger locates to the breakpoint and displays the source code.

9.16.6 Inspecting variables

Variables and channels in `ChanOutDouble()` can now be examined. For example, to examine the variable `value` press INSPECT and specify its name at the prompt. The debugger displays the value 1.0 and labels it as a `double`. Pressing INSPECT with the cursor positioned on `value` has the same effect.

Note that only variables in scope at the debugger's current location point can be inspected, although the rest of the file can be displayed with the cursor keys. The current location point is at the start of function `ChanOutDouble()`.

9.16.7 Finding addresses of variables

The debugger provides a comprehensive C expression language which may be used with INSPECT and MODIFY. To obtain the address of a variable, you use the same expression as you would in a C program. Press INSPECT and specify `&value` to display the address of `value`. Notice that addresses are displayed in hex notation by default. TOGGLE HEX may be used to display the values of variables in hex notation if required.

9.16.8 Backtracing

`ChanOutDouble()` is called from function `facs()` to output the factorial it calculates for each integer received from `feed()`. To confirm this press BACKTRACE and the debugger locates to the line in `facs()` where `ChanOutDouble()` is called. Press TOP to return to where the breakpoint occurred. Now press TOGGLE BREAK to remove the breakpoint on this line.

9.16.9 Jumping down a channel

Within `facs()` the variable `fac` is the first in a sequence of outputs on the channel `out`. To trace the destination process for `fac` first use INSPECT to see the value of the channel `out`, which is declared to be a channel pointer. Use INSPECT again but this time specify `*out`, which de-references the channel pointer. The debugger displays an `Iptr` and `Wdesc`, indicating that there is a low priority process waiting at the other end of the channel.

Now press CHANNEL and again specify `*out` to de-reference the channel pointer. The debugger jumps down the channel connecting the two processes and locates to `ChanInDouble()`. Now backtrace to the function which called `ChanInDouble()` to input a value, namely function `square()`. Variables in scope now become available for inspection (at this stage they have not been initialized).

While still in function `square()` move the cursor to the first line containing `ChanOutDouble()` and set a breakpoint. Then press RESUME in order to run the program up to the breakpoint just set.

9.16.10 Inspecting by expression

In function `square()` inspect the variable `sq` and check the computation by INSPECT and specifying the expression `x * x`. Note how INSPECT can be used to perform arithmetic on any variable in scope. Expressions can also include numbers and other variables and constants in scope at the location point.

Press INSPECT and type `x != stop_real` in order to see the value used to control the while loop.

9.16.11 Modifying a variable

In breakpoint debugging any program variable may be modified. To modify a variable `x` press MODIFY and specify `x` at the '`Destination`' prompt. The debugger now requests the new value by display the '`Source`' prompt. Enter any value and check the value has changed by inspecting `x` once again.

9.16.12 Backtracing to `main()`

While still in `square()`, press BACKTRACE to locate back to where the function was called. The debugger locates to `ProcPar()` in function `main()` where the five major processes are started in parallel. If the call to function `square()` had been nested in other calls, successive BACKTRACE operations might have been necessary but would have eventually located to the call in the program main function.

9.16.13 Entering `#include` files

Press GOTO LINE and select line 20. This will locate you to the line `#include <stdio.h>`. By using the ENTER FILE key you may now enter the `#include` file (and then any nested files within it); the EXIT FILE key will bring you out again into the enclosing file.

9.16.14 Quitting the debugger

Finally, to quit the debugger use the FINISH key (you may also quit the debugger from the Monitor page using the Q command). If the debugger was run with the '`XQ`' option, then it will prompt for confirmation before exiting.

9.17 occam debugging example

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

9.17.1 The example program

The example program `facs.occ` calculates the sum of the squares of the first *n* factorials, using a rather inefficient algorithm. It has been structured this way for clarity in process structure and to demonstrate parallel processing and debugging methods. The same program coded in C is supplied with the C toolset. The program incorporates 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 program is listed below.

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

The source file can be found in `examples/manuals/idebug`.

```
------------------------------
--
--  Debugger example:  facs.occ
--
--  Uses 5 processes to compute the sum of the squares
--  of the first N factorials using a rather inefficient
--  algorithm.
--
--  Plumbing:
--
--   feed-> facs-> square-> sum-> control <--> User IO
--     |                              |
--     -------------------------------
--
------------------------------


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

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

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

  --{{{  FUNC factorial  -  compute factorial
  REAL64 FUNCTION factorial (VAL INT n)
    REAL64 result :
    VALOF
      SEQ
        result := 1.0 (REAL64)
        SEQ i = 1 FOR n
          result := result * (REAL64 ROUND i)
      RESULT result
  :
  --}}}

  --{{{  PROC feed      -  source stream of integers
  PROC feed (CHAN OF INT in, out)
    INT n :
    SEQ
      in ? n
      SEQ i = 0 FOR n
        out ! i
      out ! stop.integer
  :
  --}}}
```

```
--{{{  PROC facs       -  generate stream of factorials
PROC facs (CHAN OF INT in, CHAN OF REAL64 out)
  INT x :
  REAL64 fac :
  SEQ
    in ? x
    WHILE x <> stop.integer
      SEQ
        fac := factorial (x)
        out ! fac
        in ? x
    out ! stop.real
:
--}}}

--{{{  PROC square     -  generate stream of squares
PROC square (CHAN OF REAL64 in, out)
  REAL64 x, sq :
  SEQ
    in ? x
    WHILE x <> stop.real
      SEQ
        sq := x * x
        out ! sq
        in ? x
    out ! stop.real
:
--}}}

--{{{  PROC sum        -  sum input
PROC sum (CHAN OF REAL64 in, out)
  REAL64 total, x :
  SEQ
    total := 0.0 (REAL64)
    in ? x
    WHILE x <> stop.real
      SEQ
        total := total + x
        in ? x
    out ! total
:
--}}}
```

```
  --{{{  PROC control    -  user interface and control
  PROC control (CHAN OF SP fs, ts,
                CHAN OF REAL64 result.in,
                CHAN OF INT n.out)
    REAL64 value :
    INT n :
    BOOL error :
    SEQ
      so.write.string.nl (fs, ts,
              Sum of the first n squares of factorials")

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

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

      n.out ! n
      result.in ? value

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

  CHAN OF REAL64 facs.to.square, square.to.sum :
  CHAN OF REAL64 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)
:
```

9.17.2 Compiling the `facs` program

The source of the program is provided in the toolset examples subdirectory. It should be compiled for transputer class TA with debugging enabled, then linked with the appropriate library files and made bootable using `icollect` with the '`T`' option to create single transputer bootable code. The example is intended for running on a B008 board wired *subs*. See section 4.7 in the accompanying *Toolset Reference Manual* debugger chapter if your system is different.

Using `imakef`

If your system has a *make* utility you may use `imakef` to generate a suitable makefile to help build the program:

```
imakef facs.bah

make -f facs.mak                          (UNIX)
make /f facs.mak                          (MS-DOS/VMS)
```

Using the tools directly

A typical sequence of commands for compiling, linking, and booting the program is shown below. The 'I' option on the linker command line is optional but does provide useful information on the progress of the linking operation.

Command sequences follow for UNIX-based and MS-DOS/VMS-based toolsets. Use the appropriate set of commands for your system.

UNIX:

```
oc -ta facs.occ -o facs.tah
ilink -ta facs.tah hostio.lib convert.lib -f occama.lnk
      -o facs.cah
icollect -t facs.cah -o facs.bah
```

MS-DOS/VMS:

```
oc /ta facs.occ /o facs.tah
ilink /ta facs.tah hostio.lib convert.lib /f occama.lnk
      /o facs.cah
icollect /t facs.cah /o facs.bah
```

9.18 Breakpoint debugging

The following section demonstrates how to debug the example program in interactive mode.

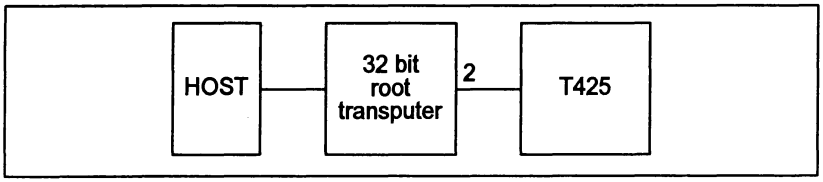


Figure 9.5 Hardware configuration for breakpoint example

9.18.1 Loading the program

The program is loaded for breakpoint debugging by running `idebug` in interactive mode using one of the commands given below. Use the appropriate command for your system.

```
idebug -sr -si -b2 facs.bah -c t425       (UNIX)
idebug /sr /si /b2 facs.bah /c t425       (MS-DOS/VMS)
```

This command starts up the debugger and displays the Monitor page but does not start the program. The **iserver** 'SI' switch is optional.

Note: If your transputer is not a T425 you should change the **t425** option to the appropriate transputer type. You may also need to change the number specified after the 'B' option to the number of the root transputer link where your network is connected. See table 4.4 in chapter 4 of the accompanying *Toolset Reference Manual* for more details about the options to use if in doubt.

9.18.2 Setting initial breakpoints

Initial breakpoints can often be set with the Monitor page $\boxed{\text{B}}$ command and specifying an entry point breakpoint (this would set a breakpoint at **facs.entry**). In this example a different method is used based on setting specific breakpoints in the source code before the program is started.

At the Monitor page select option $\boxed{\text{F}}$ to display the source file. At the object module filename prompt specify the compiled object file **facs.tah**. The debugger uses debug information within the object module to select the source file. The source file **facs.occ** is displayed with the cursor positioned at the first procedure definition, namely **facs.entry**. At this point the program is still waiting to be started.

Use $\boxed{\text{GOTO LINE}}$ to move the cursor to line 56 (**out ! fac**) and set a breakpoint there using $\boxed{\text{TOGGLE BREAK}}$. The debugger confirms the breakpoint is set and gives the breakpoint a unique identification number.

9.18.3 Starting the program

Return to the Monitor page using the $\boxed{\text{MONITOR}}$ key and start the program by selecting the $\boxed{\text{J}}$ command. Press $\boxed{\text{RETURN}}$ at the '**Command line**' prompt (no command line is required) and give a small positive number (e.g. 12) when the program prompts for input. The program runs until it reaches the breakpoint.

9.18.4 Entering the debugger

At the breakpoint the debugger displays the number of the breakpoint and the number of times it has been encountered (or hit) and then requests confirmation to continue the stopped process. Press any key except $\boxed{\text{R}}$ or $\boxed{\text{r}}$ to enter the symbolic debugging environment. The debugger locates to the breakpoint and displays the source code.

9.18.5 Inspecting variables

Variables and channels in **facs** can now be examined. For example, to examine the variable **fac** move the cursor to **fac** and press $\boxed{\text{INSPECT}}$. The debugger displays the value as **REAL64 1.0** and gives its address. Pressing $\boxed{\text{INSPECT}}$ with the cursor positioned on a space causes the debugger to prompt you for a symbol. Note that only variables in scope at the debugger's current location point can be inspected, although the rest of the file can be displayed with the cursor keys. The current location point is line 56 in the procedure **facs**.

9.18.6 Backtracing

facs is called in parallel by **facs.entry** to output the factorial it calculates for each integer received from **feed**. To confirm this press $\boxed{\text{BACKTRACE}}$ and the debugger locates to the line in **facs.entry** where **facs** is called. Press $\boxed{\text{TOP}}$ to return to where the breakpoint occurred. The current location point is line 56 in the procedure **facs**.

9.18.7 Jumping down a channel

Within **facs** the variable **fac** is the first in a sequence of outputs on the channel **out**. To trace the destination process for **fac** first $\boxed{\text{INSPECT}}$ the channel **out**. The debugger displays an **Iptr** and **Wdesc**, indicating that there is a low priority process waiting at the other end of the channel.

Now press $\boxed{\text{CHANNEL}}$ and again specify **out**. The debugger jumps down the channel connecting the two processes and locates to the corresponding channel input in procedure **square** (the statement **in ? x**). Variables in scope within **square** now become available for inspection (at this stage they have not been initialized).

9.18.8 Modifying a variable

In breakpoint debugging program variables may be modified. Start by first inspecting **x** in order to ensure that the new value will be different. To modify the variable **x** position the cursor on **x** and press $\boxed{\text{MODIFY}}$. At the modify value prompt specify the value to be placed in **x**. Note that the modify prompt reminds you of the type of **x**. Enter any valid value and check the value has changed by inspecting **x** once again.

9.18.9 Entering #INCLUDE files

Press $\boxed{\text{GOTO LINE}}$ and select line 17. This will locate you to the line **#INCLUDE "hostio.inc"**. By using the $\boxed{\text{ENTER FILE}}$ key you may now enter the **#INCLUDE** file (and any nested **#INCLUDE** files within it); the $\boxed{\text{EXIT FILE}}$ key will bring you out again into the enclosing file.

9.18.10 Resuming the program

To resume execution of the program from the current breakpoint press the $\boxed{\text{RESUME}}$ key. This will cause the program to continue running until it encounters the breakpoint again. Press an appropriate key to enter the symbolic debugging environment. This will cause the debugger to locate to line 56.

9.18.11 Clearing a breakpoint

To clear the breakpoint already set at line 56 use the TOGGLE BREAK key. The debugger will confirm that the breakpoint has been cleared. Press RESUME to resume execution and cause the program to display its result. The debugger will confirm that the program has finished and will pause in order to enable you to read the output from the program. Press any key as indicated to enter the Monitor page. Note that the Monitor page displays the exit status from the program.

9.18.12 Quitting the debugger

Finally, to quit the debugger you can use the Monitor page Q command. You may also quit the debugger from symbolic mode by using the FINISH key. If the debugger was run with the '`XQ`' option, then it will prompt for confirmation before exiting.

9.19 Post-mortem debugging

The following section demonstrates how to debug the example `facs` program in post-mortem mode.

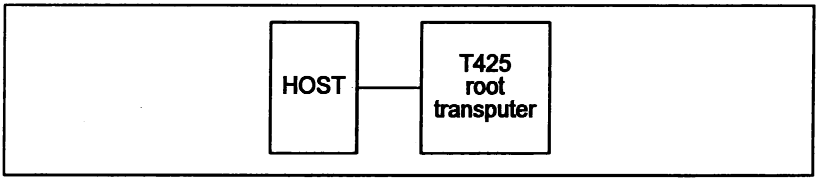


Figure 9.6 Hardware configuration for post-mortem example

9.19.1 Running the example program

When you have built an executable code file you can run the program by typing one of the following commands:

```
iserver -se -sb facs.bah                (UNIX)
iserver /se /sb facs.bah                (MS-DOS/VMS)
```

The program immediately prompts you for a value. For correct execution the number must be less than 100. To create an error for the purpose of this example, enter the value 101 and press RETURN. The program will fail with the message:

```
Error - iserver - Error flag raised by transputer.
```

9.19.2 Creating a memory dump file

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

```
        idump facs 15000
```

This creates a file called `facs.dmp` containing the transputer's register contents and the first 15000 bytes of memory. You are then returned to the operating system prompt.

9.19.3 Running the debugger

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

```
        idebug -si facs.bah -r facs -c t425        (UNIX)
        idebug /si facs.bah /r facs /c t425        (MS-DOS/VMS)
```

The `iserver` 'SI' switch is optional. The 'R' option identifies the program as one that was executed on the *root* transputer and specifies the memory dump file to be read.

Note: If your transputer is not a T425 you should change the `t425` option to the appropriate transputer type.

Should you wish to run the debugger a second time on this single processor example, without an intervening `idump` command, you will need to add the `iserver` 'SR' option to the command line to reset the network.

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 4 of the accompanying *Toolset Reference Manual*.

Inspecting variables

When the debugger is displaying source code, you may inspect any variable by placing the cursor on the variable and pressing $\boxed{\text{INSPECT}}$.

For example, to display the value of **x**, place the cursor over **x** in the source code and press $\boxed{\text{INSPECT}}$. **x** is displayed in both decimal and hexadecimal forms, and its address in memory is given in hexadecimal. For example:

```
  REAL64 'x' has value ...
  9.3326215443944096E+155 (#605166C698CF1838) (at #80000464)
```

In the same way you can inspect the values of **sq**, **square**, **stop.integer**, **stop.real**, and any other variable or constant that is in scope. Use the cursor keys to scroll through the code. To return to the source of the original error, use the RELOCATE function. You can also use the INSPECT function to examine procedures and functions. If you place the cursor on a procedure or function name and press INSPECT, the debugger displays its address and workspace requirements. You can also examine any symbol in the source by specifying its name. To do this, move the cursor to a blank area and press INSPECT. The debugger then prompts for the symbol name.

Inspecting channels

The debugger can also examine processes on channels within the scope of the original error. If you place the cursor on channel **out** and press INSPECT, information about the channel is displayed. For example:

```
CHAN 'out' has Iptr:#800022F8 and Wdesc:#80000381 (Lo) (at #8000063C)
```

This indicates that there is a process waiting for communication on channel **out**, and that it is a low priority process. To find out which occam process is waiting, press CHANNEL. The cursor will be placed on the line corresponding to the other process, which in this example is inside the procedure **sum**, on the following line:

```
        in ? x
```

Within procedure **sum**, you can examine any symbol using INSPECT. Within the **sum** procedure you can inspect the channel **out** and use CHANNEL to jump to the waiting process, which is the procedure **control** that is waiting for the final result. Again you can use INSPECT to examine any symbol.

Retracing and Backtracing

So far the debugger has located three of the five processes that compose the program. What about the others? First use the RETRACE key to retrace your steps back to procedure **square**. When in procedure **square**, inspect channel **in**, which is connected to the **facs** procedure. It is empty, which means that no process is waiting to communicate.

Next try BACKTRACE. This function backtraces down nested procedure calls. Each time the function is used the debugger locates to 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 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:

```
Goto process - use [CURSOR] then [RETURN], or 0 to F, (I)ptr, (L)o or
(Q)uit
```

To display the first active process[1], 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.

1. For a full explanation of the possible responses see the definition of the Goto Process command in the `idebug` reference chapter (chapter 4 of the accompanying *Toolset Reference Manual*).

Advanced techniques

10 Advanced use of the configurer

This chapter describes how to use the advanced features of the configurer. It is aimed at users who wish to override certain configuration defaults. The chapter deals with two topics:

- Memory usage by the configurer (code placement)
- Channel communications (channel placement and routing).

These allow the user to override the default allocation of user's code and data in memory, and to refine the channel communication for the target network using advanced virtual routing techniques. An example configuration using virtual routing is provided at the end of the chapter.

10.1 Code and data placement

The configuration language recognizes seven processor attributes ('`reserved`', three '`location`' attributes, and three '`order`' attributes), which influence the use of memory. These are described, with examples, in sections 6.5.5 to 6.5.7. This section describes the circumstances in which these attributes should be used.

'`location`' and '`order`' attributes are normally disabled and must be explicitly enabled by the configurer '`RE`' option. **Note:** When these attributes are enabled, debugging using the toolset debugger `idebug` is not supported.

10.1.1 Default memory map

By default, code is mapped into memory in the following order beginning at **LoadStart**: workspace; code; vector space. The memory segments are contiguous. The upper limit of the memory available to the configurer is defined by the `memsize` attribute specified for the processor nodes.

By default, the configurer only knows about this continuous block of memory, whose upper and lower limits are set by the value of `memsize` minus the **LoadStart** offset for the processor. The default memory map is illustrated in Figure 10.1.

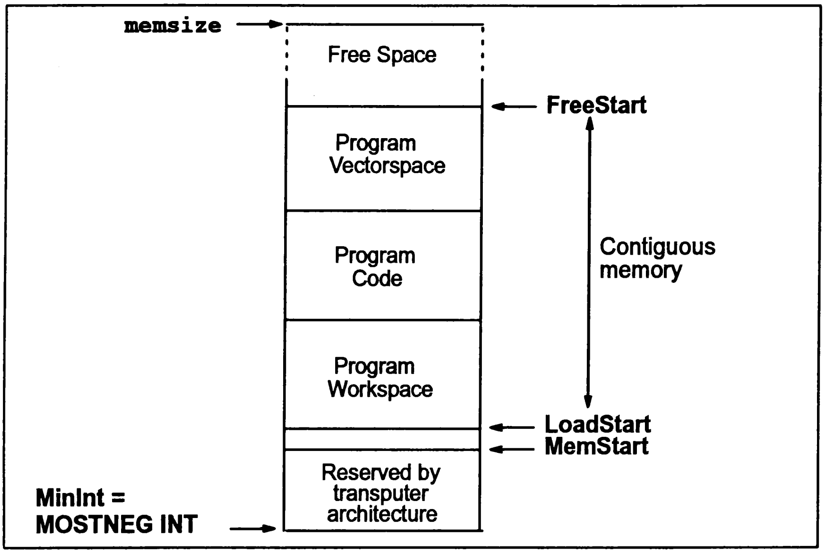


Figure 10.1 `occonf` default memory map

The first 2 or 4Kbytes of memory above `MOSTNEG INT` is implemented as on-chip RAM, and includes a few words which are reserved by the transputer hardware for the implementation of links and other hardware registers. **LoadStart** is either just above or coincident with **MemStart**.

10.1.2 Other memory configurations

Figure 10.2 illustrates a memory configuration with additional requirements to those provided by the configurer in default mode. To cater for such situations the `reserved` and `location....` attributes are supported by the configuration language.

Figure 10.2 illustrates two different sets of possible requirements:

- The first is where the available memory is discontinuous and the lowest block of memory is not sufficiently large enough to hold all the code and data.
- The second is where a block of memory is available outside the default range of memory addressed by the configurer (see above).

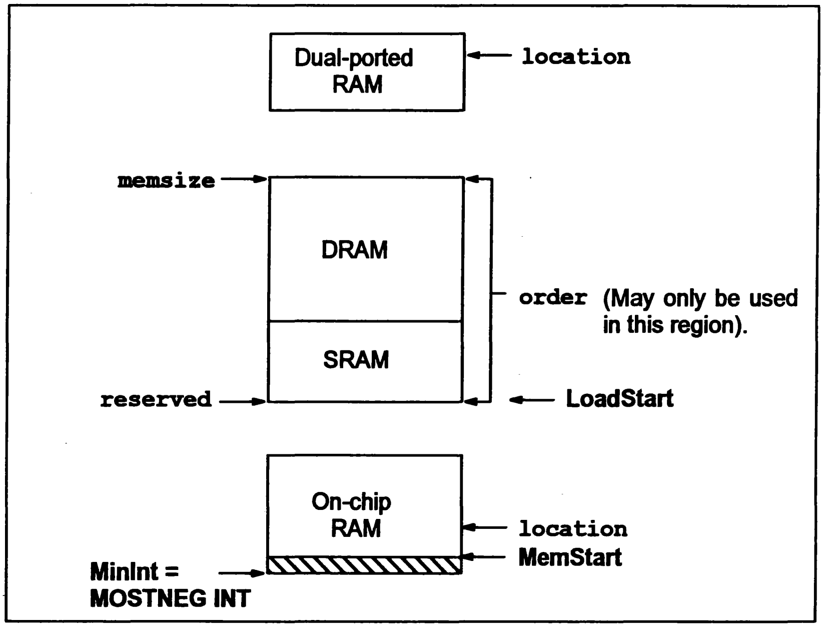


Figure 10.2 Example discontinuous memory map

10.1.3 `reserved` attribute

This attribute is used to specify the size of memory, in bytes, to reserve from `MOSTNEG INT` which cannot be used by the configurer to place user and system processes. The programmer may then use this reserved block in any way, for example, to place code and data segments of specific user processes into reserved memory using the `location`.... attributes. For example, in Figure 10.2 the `reserved` attribute has been used to force the configurer to place system and user code into the second block of memory and to ignore the on-chip RAM.

Checks are performed to ensure that the `reserved` memory size is greater than the default LoadStart offset for the processor and less than the memory size specified by the `memsize` attribute. The configurer will also ensure that the size is word aligned by rounding the size up to the nearest word boundary. **Note:** the value of the default LoadStart is variable.

When the `reserved` attribute is used, the region of memory available to the configurer for automatically placing the non-addressed code and data segments of system and user processes is defined as being:

> the top of memory as specified by the `memsize` attribute minus the memory size specified by the `reserved` attribute.

If no **reserved** attribute is defined then the region of memory available to the configurer is:

> the top of memory as specified by the **memsize** attribute minus the default **LoadStart** offset for the processor.

The **reserved** attribute is set within the configuration file **MAPPING** section using a physical processor name.

Example:

```
MAPPING
  DO
    SET processorname (reserved := 5*1024)
:
```

10.1.4 location attributes

There are three attributes which allow absolute addresses to be optionally specified for the code and data segments of a process: **location.ws**; **location.vs**; and **location.code**, corresponding to occam workspace, vectorspace, and program code respectively. As an example, Figure 10.2 indicates how the location attributes can be used to access memory below **LoadStart** (which has been changed from its default value by the **reserved** attributes) or spare memory locations available on external RAM.

Note: **location....** attributes override the equivalent **order....** attributes if specified.

Checks are performed to ensure that any code and data segments that have been absolutely addressed using the **location....** attributes are not placed into an illegal region of memory, such as:

- memory used by the configurer for automatically placing code and data segments i.e. the region defined by **Loadstart** and the **memsize** attribute. (See section 2.12.1 in the *occam 2 Toolset Reference Manual*).
- address locations that exceed the highest possible memory address location for the processor.

The configurer will fail with an error message if either of the above occur. An error will also be generated if the addresses specified are not word aligned.

A further check is made that the addresses are non-overlapping and a warning will be generated if they are. It is not illegal to have overlapping regions of memory within the permitted regions for configuration code, as described above. However, it is the programmer's responsibility to ensure there is no conflict in the use of overlapping regions at runtime.

A warning will also be generated if the **location....** attributes place code or data at address locations that exist below **MemStart**.

If the `location....` attributes are not specified then the configurer will automatically place non-addressed code and data segments.

The `location....` attributes are set on a physical processor name within the configuration file `MAPPING` section.

Example (on a 32-bit processor):

```
MAPPING
  DO
    SET processorname (location.code := #80000100)
:
```

This example specifies the start address for the process code segment. It assumes that **LoadStart** has been redefined, using the `reserved` attribute.

10.1.5 `order` attributes

The three `order` attributes described in section 6.5.5, can be used in conjunction with the `reserved` and `location....` attributes. The `order....` attributes are used to change the ordering priority of those process segments automatically placed by the configurer i.e. non-addressed code and data segments. They only operate within the memory region delimited by **LoadStart** and the value of the `memsize` attribute.

10.1.6 `location` versus `order` attributes

`location.code`, `location.ws` and `location.vs` attributes act on the same parameters as `order.code`, `order.ws` and `order.vs`, namely, program code, workspace and vectorspace.

As stated in section 10.1.4, if both the `location....`and `order` attributes are specified for a particular segment, e.g. vectorspace, then the `location....` attribute will override the effect of the `order....` attribute.

10.2 Channel communication – configuration techniques

When software virtual routing is required, the configurer works by adding multiplexing and demultiplexing processes to implement a number of virtual channels over a single hardware link. It will also add routing processes to through-route data between processors which are not directly connected. In doing so it assumes by default that:

- any link to link connections in the target network can be used for implementing virtual channel traffic.

- any of the processors can be used for through-routing.

- where multiple routes of the same length exist between two processors, the virtual channels between these processors should be shared out between these routes as much as possible.

While these are, in general, reasonable assumptions, users may require more control over how processors and links are used for implementing virtual channels in specific networks. The configurer permits users to control its routing decisions by means of processor attributes and channel placements which can be defined in the configuration source file. These are designed to supply the following capabilities:

- A channel may be placed on a specific hardware `arc` between processors. This instructs the configurer to implement the channel directly using the hardware link rather than as a virtual channel. Only two channels may be placed (one in each direction) on a hardware link. This can be used to ensure that a limited number of critical channels are directly implemented by hardware links. **Note:** that this placement is ignored if both interactive debugging (with `idebug`) and virtual routing are enabled.
- It is possible to prevent specific processors from being used as pathways for virtual channels required by other processors. This ensures that certain critical processors within the target system are not used for through-routing virtual channels for less critical processors.
- It is possible to ensure that all virtual channels are routed via a group of processors specifically placed in the target network to support them. Hence a group of small inexpensive processors may be placed in the middle of a network of processors to provide the communications requirements at little cost to the other processors.
- It is possible to control the number of virtual channel support processes that are added to particular processors, and also whether they are given use of internal memory in preference to application processes. This preserves the performance of critical processors in the target network and allows virtual channel support on processors with limited memory capacity.

The following sections describe the use of the `PLACE` statement and the `order....` attributes to optimize important channels and to make the best use of fast memory. Section 10.2.4 introduces the additional attributes used to control the configurer's routing system and describes how to use them to meet the requirements identified above. An example is described in section 10.3.

10.2.1 Routing and placement constants

The include file `occonf.inc` contains a number of constants associated with the routing and placement attributes. The file should be referenced at the top of the configuration before the hardware description if any of the configuration constants mentioned in the following sections are used, e.g.

```
#INCLUDE occonf.inc
```

10.2.2 Optimizing important application channels

By placing an application channel on a hardware `arc` it is possible to reserve the hardware link solely for the use of the application channel concerned (except when interactive debugging with `idebug` and virtual routing are both enabled).

With this technique a sub-set of the channels used by an application can be placed on a sub-set of the hardware links available within the target system. This then optimizes the performance of the placed data paths.

When doing the placement the user must be careful to leave at least enough free links to form a minimal spanning tree between each sub-set of processors in the target network that require through-routed virtual channels to connect them. (See section 10.2.4).

10.2.3 Virtual communications – use of fast memory

Normally the workspace segment of virtual channel support processes (added to the target network by the configurer), where used, is allocated within fast memory (i.e. at the most negative addresses) before the user process code and data segments are allocated.

User process code and data segments can, however, be allocated from internal store before the stack of the virtual channel support processes is allocated. This is done by setting `order` attributes for the relevant user processes to lower values than those automatically given to the stack segments of the virtual channel support processes.

Virtual channel support processes are divided into *routing* processes and *multiplexing/demultiplexing* processes. Workspace segments of all *routing* processes placed by the configurer are all given the value `ROUTER.ORDER`. Workspace segments of all *multiplexing/demultiplexing* processes placed by the configurer are given the value `MUXER.ORDER`. Default values of `-20000` and `-10000` respectively are defined for these constants in the include file `occonf.inc` which is supplied with the toolset.

So, if `order` values on the code and data segments of user processes are less than `ROUTER.ORDER` the segments concerned will be allocated from internal store before *any* of the virtual channel support processes' workspace is allocated.

If `order` values on user processes are less than `MUXER.ORDER` but greater than `ROUTER.ORDER`, only the workspace required by *routing* processes will be allocated before the configurer allocates space for the user processes concerned.

Caution: If the stack segments of heavily-used virtual channel support processes are pushed out of internal store by giving priority to user processes, the impact on the performance of the virtual links and the processor will be quite noticeable. User processes should only be given priority over the virtual channel support processes on a processor if the amount of data through-routed by the processor during normal operation is likely to be small.

Giving user processes priority use of fast memory will only impact the performance of those virtual channels used by processes on the processor. The CPU cost of supporting those virtual channels will only be slightly increased.

10.2.4 Control of routing and placement

This section describes how the allocation of a virtual routing system across a network can be controlled. For example, particular routes can be avoided or promoted as required.

Introduction to routing and placement attributes

User control of routing and placement is performed by means of three extra processor attributes `routecost`, `tolerance`, and `linkquota`. These are specified in the `MAPPING` section of the configuration file. They are specified for physical processor names using the following syntax:

```
SET processorname (routecost := exp)

SET processorname (tolerance := exp)

SET processorname (linkquota := exp)
```

Routing cost

`routecost` can be used to make the configurer choose one processor over another when deciding how to route channels in the network. In the default case, all processors and links in the network are assumed to be equally usable. When deciding how to route a channel between two processors, the configurer works out the routes between the two points, and then calculates the "cost" of each route by counting the number of processors on each route. The "best" of these (the one with the least number of processors) is then chosen to implement the channel, and the appropriate through-routing processes are placed on each intermediate processor on the route. If there are a number of channels to be implemented between the two ends, and there is more than one route of the same ("best") length available, then the channels are shared between the available routes.

`routecost` allows a *routing cost* to be explicitly allocated to one or more processors in the network. The cost of a route between two processors is then determined not simply by the number of intermediate processors, but by the *sum of the routing costs* of all the intermediate processors. There is a default routing cost for processors which have not had one explicitly allocated. So by giving a *high* routing cost value to a processor, this will discourage the configurer from using it as an intermediate node when routing channels. Similarly by giving it a *low* cost compared with other processors in the network, this will encourage the configurer to use it for through-routing.

A value greater than or equal to the maximum permitted value `INFINITE.COST` (defined in `occonf.inc`) prohibits through-routing on that processor.

Tolerance

The second attribute – `tolerance` – controls how the configurer decides to share out channels between available routes. If there are a number of channels to be implemented between two processors, then the configurer normally calculates the cost of each possible route, and then shares out the channels between available "best" routes with the least cost. If there is only one "best" route then all the channels will go via that one. In some circumstances it may be better to share out the channels more evenly, to prevent bottlenecks in the system, even if this results in some channels being implemented on slightly higher cost routes. The `tolerance` attribute for a processor is designed to allow this.

When calculating whether to use a route for channel sharing, the configurer uses the minimum of the `tolerance` values of the processors on that route. It subtracts that tolerance from the route cost; if the result is less than the cost of the "best" route, then this route, as well as the "best" routes, may be used for load-sharing of channels. As an example, consider a network in which all processors have been given the same routing cost (say 1000). Normally, this would result in load-sharing of channels only when the routes are the same length. However, if the tolerance of all the processors were set to twice the routing cost value (2000), then the configurer would also include routes with one more processor on them than the "best" route for channel load-sharing.

When setting up a network, the `routecost` attributes should be set first to indicate which processors are preferred for through-routing. Then the `tolerance` attribute can be set, for all processors in the network, to influence the load-sharing strategy. In general a set of processors in a network (or in part of a network) would be given the same `tolerance` value to indicate the load-sharing strategy required for that network (or part of the network). The likely cases are:

- A zero `tolerance` value indicates that virtual channels should only be placed on a route if it is the *only* "best" route between two processors. If all "best" routes have zero tolerance, then one will be picked arbitrarily and *all* virtual channels will be routed on that one.

- A default `tolerance` value indicates that channels may be shared between the "best" routes between two processors.

- A `tolerance` value which is some multiple of the routing cost values in the network indicates that channels should be shared between the "best" routes and those routes with a higher cost but with tolerance values indicating that they are also acceptable.

- The maximum `tolerance` value indicates that *all* routes between two processors can be used for channels. This might lead to some very long routes being chosen.

`ZERO.TOLERANCE`, `DEFAULT.TOLERANCE` and `MAX.TOLERANCE` are defined in the include file `occonf.inc`.

Link quota

The third attribute – `linkquota` – controls how many links on a processor may be used to carry virtual channels to the processes on that processor. In the default case any of the four links may be used. For each link which is used, a small additional memory overhead is incurred. On processors with very small amounts of memory it may be important to keep the memory overhead as low as possible.

The `linkquota` attribute can be set to a value in the range 0 to 4 inclusive. It should only be set to 0 if no virtual channels will be required by the processes on that processor. If it is set to 1, then the processes on the processor may use virtual channels, but it should be possible for the configurer to implement them all via *one* of the processor's links. Similarly for values of 2, 3, and 4 (although, obviously, setting the quota to 4 on a processor with four links has no effect).

The `linkquota` attribute is a *guide* to the configurer rather than an absolute directive. If a processor has a `linkquota` value of 1, but the processor provides the only route available for the implementation of a particular channel in the network, then the configurer will choose to route data through that processor, even though this will cause the link quota to be exceeded.

The `linkquota` is not intended as a method of avoiding routing through a processor; the `routecost` attribute should be used for that. Instead it is intended to indicate, on memory-critical processors, that the minimum overhead should be placed on them. The quota should reflect the requirements of the processes placed on that processor, and the routing costs in the network should be chosen so that other processors are used for through-routing. The link quotas will then be checked by the configurer as it sets up the multiplexing and routing processes. The configurer will output a warning message if it has exceeded a quota. The network can then be re-examined to see why this is happening.

The minimal spanning tree

There is one aspect of the implementation of virtual channels which may become evident when constraints are placed on how the configurer may route channels in the network. Normally the configurer can use any of the links in the network for virtual channels, so if the network is connected, then virtual channels can be routed from any processor to any other. However, (as described in section 10.2.2) it is possible to `PLACE` a pair of opposing channels on a link in the network; in this case the link is used directly to implement those two channels, and cannot be used for virtual channels. Also the `routecost` attribute on selected processors in the network may prevent the use of some processors (and hence links) in the network for through-routing. If too many links are removed from the network in this way then it may become impossible to implement some of the virtual channels required.

So it is important to ensure that, for a set of processors in a network requiring virtual channels to be connected between them, there is a set of links connecting the processors over which virtual channels is allowed. This set of links will then be used by the configurer to construct a *minimal spanning tree* of links to ensure that it can always implement the virtual channels between these processors. Any additional links available for virtual channels will also be used to provide better routes between processors. If the configurer is unable to construct the route necessary to implement a requested virtual channel, it will give an error message.

A network may not require a single minimal spanning tree to cover the whole network; it depends on the virtual channel requirements of the configuration. For example, it might be possible to divide a configuration into two separate parts, each requiring virtual channels internally, but with a single pair of channels (which can be directly mapped onto a link) joining the two parts. In this case a minimal spanning tree of links is required for each of the two parts. These are known as *sub-networks*.

Summary of routing and placement attributes

The attributes are defined in more detail as follows:

- `routecost` - defines, within the range `MIN.COST` to `MAX.COST` inclusive, the associated cost of routing virtual channels through a particular processor. Default values of `1` and `1000000` respectively are defined for these constants in the include file `occonf.inc`

  If a value greater than the maximum of `MAX.COST` (e.g. `INFINITE.COST`) is specified then no through-routing will be permitted on that processor.

  The default value for this attribute is 1000 i.e. `DEFAULT.COST`.

- `tolerance` - controls how much a particular processor can be used to provide load-sharing routing paths for other processors. It uses any value in the range `ZERO.TOLERANCE` = 0 to `MAX.TOLERANCE` =1000000 inclusive.

  The default value for this attribute is 1 i.e. `DEFAULT.TOLERANCE`. This allows the processor to implement alternate routes for through-routed channels with exactly the same total cost as the "best" route found between any two other processors.

  If the value `ZERO.TOLERANCE` is specified then the processor will only be used for through-routing if it lies on the "best" route found to implement virtual channels.

If the maximum value `MAX.TOLERANCE` is specified on all processors in the target network almost every possible route will be used to share the cost of carrying data between any pair of non-adjacent processors.

- `linkquota` - suggests the maximum number of links on the processor that should be used by the virtual channel routing system.

  `linkquota` can have the values 0 to 4 inclusive.

  A warning will be produced if the suggested `linkquota` for a node is exceeded. The `linkquota` will only be exceeded because of the requirements of through-routing data for other processors.

Default values for these attributes are defined in the include file `occonf.inc` which is supplied with the toolset.

Prevention of through-routing via critical processors

If there are processors within the target network that are likely to be CPU-limited by the application, then it may be undesirable to allow virtual channels from surrounding processors to be routed through the performance-critical processors. In this case the `routecost` attribute for the critical processors should be set to `INFINITE.COST`. If this is done then no virtual channels can be through-routed via these processors.

Care must be taken to ensure that a minimal spanning tree of links is provided by the other processors in the network. If a particular processor should only be used for through-routing channels when absolutely necessary, then the `routecost` attribute on the processor can be set to some multiple of the default value. Alternatively the cost value can be explicitly set on the other processors. If for example, the multiple concerned is larger than the number of lower cost processors in the network then any route via those processors will be chosen in preference to a route via one of the high cost processors.

Use of additional processors for through-routing

There may be situations when the configurer is required to route all communications via a particular set of processors. For example:

- to emulate closely the communications structure that would be provided by dedicated hardware routing devices, or
- when a block of low performance processors is provided in the target network solely for the purposes of through-routing data for other processors.

This can be achieved in one of two ways:

- If the `routecost` of all processors, other than those intended as routers, is set to `INFINITE.COST` then the only processors that the configurer can use for through-routing are those left with the default routing cost. This technique has the advantage of guaranteeing that no through-routing will be done via the standard processors.

- If the `routecost` of all the routing processors are set to a small value then any route via these processors will be used in preference to routes via processors with the default routing cost. This technique has the advantage that the normal processors can still be used by the configurer for routing channels that cannot be implemented by the nominated routing processors. Hence the nominated routing network need not provide full connectivity.

Generally the second method is preferred as it preserves the ability of the configurer of mapping an arbitrary application onto the target hardware.

Support for memory-critical systems

It may be desirable to ensure that for a particular processor the additional run-time overhead added by the configurer is kept to a minimum.

Normally the configurer spreads virtual channels running between a pair of processors across all routes that have equal cost. For each additional route employed additional support processes may be required and hence additional memory consumed on the target system.

This should not normally be a problem as the total cost of the maximal set of run-time processes that can be placed on the target system by the configurer consumes only a few thousand bytes more than the minimal set.

Some example figures of the minimum and maximum costs of both through-routing and multiplexing software on different word length transputers are shown below (all sizes are in bytes):

Word Size	Function	Code	Min Stack	Max Stack
32 bits	Through-routing	699	768	2112
	Multiplexing	1940	784	2056
16 bits	Through-routing	708	512	1568
	Multiplexing	1952	524	1556

Multiplexing software is needed whenever a processor has virtual channels terminating on it. In the current system each opposing pair of virtual channels forming a virtual link will require approximately 120 bytes of local storage on a 32-bit processor and 80 bytes of storage on a 16-bit processor.

Note: In the default configuration case, extra overheads will be incurred to allow interactive debugging of the application. Use the `Y` command line option to override this.

A particular case of the critical memory problem comes when the set of user processes on a particular processor do not in themselves require virtual channels at all, because the channels they use can be mapped directly onto the hardware links available. However, if the configurer decides to use through-routing then through-routing support processes will be added to the processor. In addition, to enable the available hardware links to be shared, some of the channels used on the processor may be implemented as virtual channels. In this case multiplexing software will also be required. In this special case the processor can be completely protected from run-time overheads by using the techniques described above under the heading *Prevention of through-routing via critical processors.*

A `linkquota` attribute can be specified on each processor in the target network. If the `linkquota` of a particular processor is specified as `1` and the `routecost` set to `INFINITE.COST`, then only a single hardware link will be used on the processor to provide all the virtual channels it uses. In addition the memory overheads of the virtual link system will be reduced to a minimum (minimal multiplexer only).

If `linkquota` is set to 1 on all processors in the target system then the minimal spanning tree of links will be used to support all virtual channels required. Warnings will be produced in this case for all processors that have had more than `linkquota` links used on them; this is because all processors cannot be chosen as "leaves" in the spanning tree.

If both performance and memory size are a problem in a particular system it is likely that the user will have to tune the `linkquota` and `tolerance` parameters of many processors in order to get the best result.

10.3 Example – optimized filter test program

Figure 10.3 describes an example configuration that needs to be placed onto a network of six processors (Figure 10.4). The function of the program is to test the two filter components which are limited by the speed of the processors concerned. Sources are supplied in the `examples/manuals/advconf` directory.

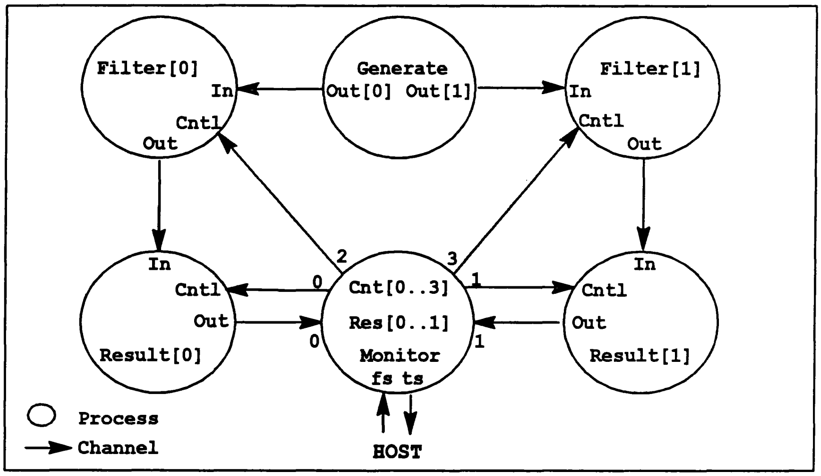


Figure 10.3 Example filter test program

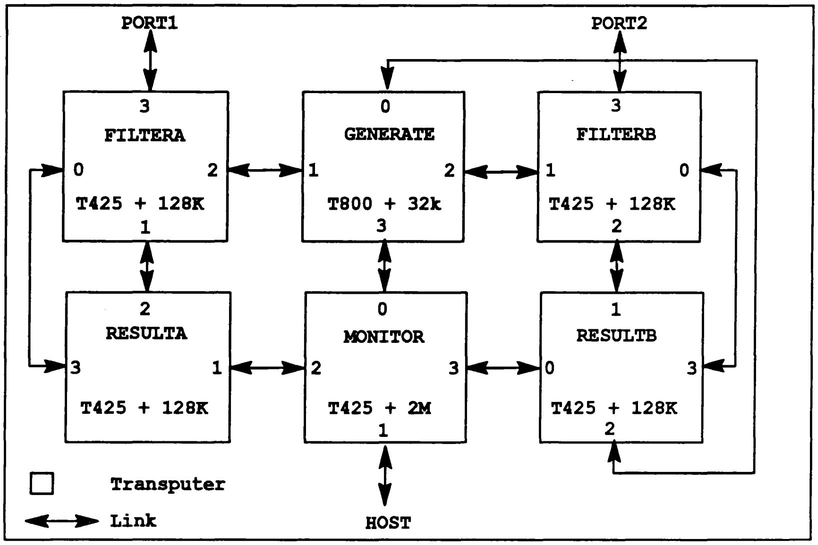


Figure 10.4 Example filter test hardware

This is not a real program but has been constructed to demonstrate many of the features for optimization described in the previous sections, within a comparatively small and simple system. The basic configuration description is listed in the following example:

```
-- Include values for router attributes
#INCLUDE "occonf.inc"

-- Hardware description for specialised sub-system

NODE GENERATE, FILTERA, FILTERB :
NODE RESULTA, RESULTB, MONITOR :
EDGE port1, port2 :
ARC hostarc :

-- The following ARCs are only required when optimising
ARC GENERATE.TO.FILTERA, GENERATE.TO.FILTERB :
ARC FILTERA.TO.RESULTA, FILTERB.TO.RESULTB :

NETWORK
  DO
    SET GENERATE (type, memsize := "T800",  32*K)
    SET FILTERA  (type, memsize := "T425", 128*K)
    SET FILTERB  (type, memsize := "T425", 128*K)
    SET RESULTA  (type, memsize := "T425", 128*K)
    SET RESULTB  (type, memsize := "T425", 128*K)
    SET MONITOR  (type, memsize := "T425",   2*M)

    CONNECT HOST              TO MONITOR[link][1] WITH hostarc
    CONNECT MONITOR[link][2]  TO RESULTA[link][1]
    CONNECT MONITOR[link][3]  TO RESULTB[link][0]
    CONNECT MONITOR[link][0]  TO GENERATE[link][3]
    CONNECT GENERATE[link][1] TO FILTERA[link][2] WITH
            GENERATE.TO.FILTERA
    CONNECT GENERATE[link][2] TO FILTERB[link][1] WITH
            GENERATE.TO.FILTERB
    CONNECT RESULTA[link][2]  TO FILTERA[link][1] WITH
            FILTERA.TO.RESULTA
    CONNECT RESULTB[link][1]  TO FILTERB[link][2] WITH
            FILTERB.TO.RESULTB
    CONNECT RESULTA[link][3]  TO FILTERA[link][0]
    CONNECT RESULTB[link][3]  TO FILTERB[link][0]

    CONNECT GENERATE[link][0] TO RESULTB[link][2]

    CONNECT FILTERA[link][3]  TO port1
    CONNECT FILTERB[link][3]  TO port2
:

-- Software description for filter test program

NODE generate.p, monitor.p :
[2]NODE result.p, filter.p :

#INCLUDE "hostio.inc"
#USE "generate.c8h"
#USE "filter.c5h"
#USE "result.c5h"
#USE "monitor.c5h"

CHAN OF SP fs, ts :
[2]CHAN OF BYTE Out :
[2]CHAN OF BYTE Filter.to.Res :
CONFIG
  [2]CHAN OF BYTE Res :
    [4]CHAN OF BYTE Cntl :
    PAR
      PROCESSOR monitor.p
        Monitor(fs, ts, Res, Cntl)
      PROCESSOR generate.p
        Generate(Out)
      PAR i = 0 FOR 2
        PAR
          PROCESSOR result.p[i]
            Result(Filter.to.Res[i], Res[i], Cntl[i])
          PROCESSOR filter.p[i]
            Filter(Out[i], Filter.to.Res[i], Cntl[i+2])
  :

  -- Mapping description
  MAPPING
    DO
      MAP generate.p  ONTO GENERATE
      MAP filter.p[0] ONTO FILTERA
      MAP filter.p[1] ONTO FILTERB
      MAP result.p[0] ONTO RESULTA
      MAP result.p[1] ONTO RESULTB
      MAP monitor.p   ONTO MONITOR

      MAP fs, ts ONTO hostarc

      -- Mapping optimisation:

      -- Prevent through routing via GENERATE
      SET GENERATE (routecost := INFINITE.COST)

      -- Ensure minimum overhead on FILTERA
      SET FILTERA (routecost, linkquota := INFINITE.COST, 1)

      -- Ensure minimum overhead on FILTERB
      SET FILTERB (routecost, linkquota := INFINITE.COST, 1)

      -- Optimise Generate to Filter 0 Path
      MAP Out[0] ONTO GENERATE.TO.FILTERA

      -- Optimise Generate to Filter 1 Path
      MAP Out[1] ONTO GENERATE.TO.FILTERB

      -- Optimise Filter to Result 0 Path
      MAP Filter.to.Res[0] ONTO FILTERA.TO.RESULTA

      -- Optimise Filter to Result 1 Path
      MAP Filter.to.Res[1] ONTO FILTERB.TO.RESULTB

      -- Use otherwise unspecified linkquotas to check
      -- overheads on GENERATE, RESULTA, and RESULTB
      SET GENERATE (linkquota := 0)
      SET RESULTA  (linkquota := 2)
      SET RESULTB  (linkquota := 2)
  :
```

For this real-time program to actually work correctly a number of optimization features of the configurer have been exploited to ensure the right routing decisions are made:

- GENERATE has no memory space available to carry the overheads of routing software and requires no virtual channels itself, so setting `route-cost` to `INFINITE.COST` prevents routing software being placed on it.
- FILTERA and FILTERB must be operated in a state as close as possible to the real case, where all their channels are placed onto hardware links. The main data path through the `Filter` component must operate at hardware data rates, so the `In` and `Out` channels must both be placed onto hardware links to guarantee the required performance. The `Cntl` channel which carries a small amount of parameterization data can, however, be implemented as a virtual channel without significant effect.

These are implemented in the Mapping description.

Note: the example should be built with interactive debugging disabled so that explicit channel mapping can take effect.

11 Mixed language programming

This chapter describes the mechanisms for mixing code modules written in different high level languages. It is divided into two parts. The first part discusses how to call procedures and functions written in one language from another language. This includes details of the library procedures provided to allow occam programs to call C functions which require use of static or heap memory.

The second part describes how complete C programs can be called as if they were occam processes with a standard channel interface.

11.1 Mixed language programs

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 application modules may already exist in particular languages. In either case a well defined mechanism for mixing languages within a single system is desirable.

The toolset provides a clean and simple basis for mixing languages on transputer networks. Independent software processes can be written in different languages, compiled and linked using a common set of tools, and the linked modules placed anywhere on a network of transputers using a configuration description. Compiler pragmas are provided to allow code to be imported with the correct calling conventions, and to translate names so they are valid in the calling language.

Code written in other languages can be used as external routines in a program, providing the language calling conventions are honored, and no conflicts of name occur.

There are a number of issues to be considered when mixing languages. These are:

- The declaration of the external routine — in order for the calling program to be able to correctly call an external routine, it must have a description of the interface to the routine. The way in which this is done depends on the language being used.

- The translation of names — programming languages differ in the legal character set for identifiers and symbolic names. Thus, names acceptable in one language may not be valid in another. To avoid these problems compiler pragmas are provided to perform name translations.

- The calling conventions of the languages — including passing the address of the static area and the types of the parameters in the two languages.
- The types returned by functions.
- The presence, or otherwise, of a *static area* in each language (this is discussed in more detail below).
- The libraries to be used when linking the complete program.

These issues are discussed in more detail in the the following sections.

Note: When mixing languages, the external procedures must not do any host communications. All i/o should be performed by the calling program. The external procedures *can* however perform channel communications with other processes.

11.1.1 Declaring external routines

In order to properly call a separately compiled procedure or function, the compiler needs to be given information about the external routine. In C this is done by declaring the function as external, for example:

```
extern int f (int a, int b);
extern void p1 (char c);
```

The functions should be declared as prototypes, including the types of parameters, to ensure that the actual parameters are converted to the specified types. If the functions are declared without the parameter types then the default C argument type promotions will take place.

The occam compiler uses a pragma to provide information about external procedures and functions. The syntax of this is:

#PRAGMA EXTERNAL "*formal declaration* = *workspace* [, *vectorspace*]"

The optional parameter *vectorspace* is not required for C functions.

For example:

```
#PRAGMA EXTERNAL "PROC p1 (VAL BYTE c) = 20"
#PRAGMA EXTERNAL "PROC p2 (BYTE x, y) = 40, 100"
#PRAGMA EXTERNAL "INT FUNCTION f (VAL INT a, b) = 50"
```

A void function in C is equivalent to a procedure in occam.

11.1.2 Translating identifiers

Because the syntax of valid identifiers can vary from one language to another, compiler pragmas are provided in C and occam to allow the names used in a source file to differ from those used externally.

The pragma can be used to change the name which is used in the object code to reference an external routine. For example, a C program which needs to call an occam function called '`get.next`' could use the following to convert the name into a valid C identifier:

```
#pragma IMS_translate(get_next, "get.next")

extern void get_next(int *n, Channel *in);
```

Alternatively the pragma could be used to change the name 'exported' from the occam code:

```
#PRAGMA TRANSLATE(get.next, "get_next")

PROC get.next (INT next, CHAN input)
  ⋮
:
```

In this case, the object file will contain the name '`get_next`' and the procedure can only be called by this name.

11.1.3 Parameter passing

The two issues in passing parameters between languages are, firstly, the types of the formal and actual parameters (including whether they are passed by *value* or by *reference*) and, secondly, the use of a static area by each language. These are described in more detail below.

Parameter compatibility

Correct parameter passing depends on the compatibility of data types between languages. See the language implementation chapters of the appropriate *Language and Libraries Reference Manual* for details of the implementation of types and how parameters are passed.

The way in which parameters are passed — either as a copy of the data (by *value*) or a pointer to the data (by *reference*) — involves two issues: the semantics of the language, and the actual implementation.

> **C:** All parameters are passed by value. Arrays are passed as pointers to the base type of the array. It is possible to pass pointers to variables which gives the effect of passing by reference.
>
> **occam:** parameters are either VAL parameters or non-VAL parameters. VAL parameters may be implemented by passing by value, or by passing a pointer. The latter will happen when the size of the parameter is larger than the word length of the processor and will therefore depend on the data type and the processor type.

Types can be considered to be compatible if they have the same interpretation, are the same size and are passed in the same way. For example, a C parameter of type `int` is compatible with an occam `VAL INT` parameter. Similarly, as an occam `INT` parameter is passed as a pointer it is compatible with a C `int *` parameter.

When passing parameters the correct data type should be used. Equivalences for the main C and occam data types are listed in tables 11.1 and 11.2.

occam type	C type

`VAL BYTE`	`char`
	`unsigned char`

`BYTE`	`char *`
	`unsigned char *`

`VAL INT16`	`short int`
`INT16`	`short int *`
`VAL INT`	`int`
`INT`	`int *`
`INT32`	`long int *`
`REAL32`	`float *`

`VAL REAL64`
`REAL64`
	`double *`

`CHAN`	`Channel *`
`TIMER`	No parameter required

Table 11.1 Type equivalents for all processors

occam type	C type	
	16 bit processor	32 bit processor
`VAL INT32`	`long int *`	`long int`
`VAL REAL32`	`float *`	`float`

Table 11.2 Type equivalents dependent on processor word length

Comprehensive equivalence tables, with examples of calling external routines from each language, can be found in Appendix B.

Range checking in occam

It is important to ensure that parameters passed to occam procedures and functions from C have values within the legal range for the type. For example, when passing to a formal parameter of type `BYTE` the value must be in the range 0 through 255. Violation of this rule is liable to cause a runtime range check error in the occam code.

occam timers

An occam `TIMER` parameter should have *no* associated actual parameter. For example, consider the following occam procedure :

```
PROC p (VAL INT p1, TIMER t, VAL INT p2)
  SEQ
    ...
```

The C code to call the above is as follows:

```
void p(int p1, int p2);
#pragma IMS_nolink(p)
...
int x, y;
p(x, y);
```

11.1.4 Passing array parameters

In both C and occam an array parameter is passed as a pointer to the start of the array, i.e. the address of the first element. occam also supports unsized array parameters where some or all of the array bounds may be omitted from the parameter declaration. In this case the address of the array is followed by a sequence of integer parameters, one for each unknown bound, giving the value of that bound. The unknown bound parameters appear in the same order as the unknown bounds in the array parameter declaration.

In the following sections occam procedures are used in the examples. The principles described apply equally to occam functions except that an occam function may only have `VAL` parameters.

C calling occam

There are four cases to consider when calling occam routines, which accept arrays as parameters, from C. In the following examples we assume that the C declaration of the occam routine has the `nolink` pragma applied to it so that the hidden Global Static Base (GSB) parameter is not passed when we call the occam routine (see section 11.1.6). Although the examples use `INT` arrays, the same principles apply to an array of any other occam type.

1 Sized array:

```
PROC f([8]INT a)
```

To call the above from C we can declare the occam procedure as a C prototype in any of the following ways:

```
a) void f(int a[8]);
b) void f(int a[]);
c) void f(int *a);
```

The function is called as follows:

```
int a[8];

f(a);
```

2 Sized VAL array:

```
PROC f(VAL [8]INT a)
```

This is similar to case 1 except that since the array is a `VAL` array we can declare the occam routine as a C prototype which accepts a `const` array.

```
a) void f(const int a[8]);
b) void f(const int a[]);
c) void f(const int *a);
```

The function is called as follows:

```
int a[8];

f(a);
```

3 Unsized array:

```
PROC f([]INT a)
```

Here the occam procedure expects a hidden integer parameter following the array which gives the number of elements in the array. Thus we can declare this occam routine as a C prototype as follows:

```
a) void f(int a[], const int size);
b) void f(int *a, const int size);
```

The function is called as follows:

```
int a[8];

f(a, 8);
```

4 Unsized `VAL` array:

```
PROC f(VAL []INT a)
```

This is similar to case 1 except that since the array is a `VAL` array we can declare the occam routine as a C prototype which accepts a `const` array.

```
a) void f(const int a[], const int size);
b) void f(const int *a, const int size);
```

The function is called as follows:

```
int a[8];

f(a, 8);
```

Multi-dimensional arrays (C calling occam):

Multi-dimensional arrays are treated in the same way as that described for unitary arrays. The hidden array dimensions are passed in the same order as they appear in the array definition. For example, consider the following occam routine which is to be called from C:

```
PROC f([8][][]INT a)
```

This can be declared as the following C prototype:

```
void f(int a[8][][], const int bound1, const int bound2);

#pragma IMS_nolink(f)
```

Note that even though the array has three dimensions we only declare explicit extra parameters for those dimensions that are hidden.

This function can be called as follows:

```
int a[8][9][4];

f(a, 9, 4);
```

occam calling C

There are a number of cases to consider when calling C routines, which accept arrays as parameters, from occam. In the following examples we assume that the C functions to be called have been declared using the `nolink` pragma so that we do not need to pass a hidden GSB parameter (see section 11.1.6). Although the examples use `int` arrays, the same principles apply to an array of any other C type.

1 Simple arrays and pointers

```
a) void f(int a[8]);
b) void f(int a[]);
c) void f(int *a);
```

These would be declared as an occam procedure and called as follows:

```
#PRAGMA EXTERNAL "PROC f([8]INT a)=ws"

[8]INT a:
f(a)
```

Note that b) and c) cannot be declared as accepting unsized arrays in occam because they are not expecting the hidden parameters that occam would pass implicitly when `f` was called.

2 `const` arrays and pointers

```
a) void f(const int a[8]);
b) void f(const int a[]);
c) void f(const int *a);
```

These would be declared as an occam procedure and called as follows:

```
#PRAGMA EXTERNAL "PROC f(VAL [8]INT a)=ws"

[8]INT a:
f(a)
```

Note that b) and c) cannot be declared as accepting unsized arrays in occam because they are not expecting the hidden parameters that occam would pass implicitly when `f` was called.

3 Arrays and pointers accompanied by size values

It may be that the C function to be called is written in such a way that it expects an integer to follow the array which gives the number of elements in that array. This matches the parameter passing conventions for occam unsized arrays. Thus if the C function is defined as follows:

```
a) void f(int a[], const int s);
b) void f(int *a, const int s);
```

then the equivalent occam declaration and call is:

```
#PRAGMA EXTERNAL "PROC f([]INT a)=ws"

[8]INT a:
f(a)
```

When `f` is called occam implicitly passes the array bound, `8`, which is picked up as `s` by the C function.

4 `const` arrays and pointers accompanied by size values

This is similar to the above but the array in the occam declaration of the C function is now declared as a `VAL` array. Thus given the following:

```
a) void f(const int a[], const int s);
b) void f(const int *a, const int s);
```

then the equivalent occam declaration and call is:

```
#PRAGMA EXTERNAL "PROC f(VAL []INT a)=ws"

[8]INT a:
f(a)
```

Multi-dimensional arrays (occam calling C)

Multi-dimensional arrays are treated in the same way as unitary arrays. For example, consider the following C routine which we want to call from occam:

```
void f(int a[3][4]);
```

then the equivalent occam declaration and call is:

```
#PRAGMA EXTERNAL "PROC f([3][4]INT a)=ws"

[3][4]INT a:
f(a)
```

occam expects any hidden array dimensions to be passed in the same order as they appear in the array definition. Consider the following C routine, which expects the array bounds to be passed separately, and which we want to call from occam:

```
void f(int *a, const int bound1, const int bound2);
```

The equivalent occam declaration and call is:

```
#PRAGMA EXTERNAL "PROC f([][]INT a)=ws"

[3][4]INT a:
f(a)
```

When `f` is called in this case the C function will receive 3 for `bound1` and 4 for `bound2`. The bounds are passed implicitly by occam.

11.1.5 Function return values

When functions are being called it is also necessary for the return types to be compatible.

The definition of compatibility for function return types is stricter than that for parameters. Floating point and integer function results are returned in different ways (depending on the processor type) and so it is essential to ensure that the types of function return values are strictly equivalent. A partial list of equivalents is given in table 11.3 for guidance. Comprehensive tables of equivalent types can be found in Appendix B.

occam function type	C function type

`BYTE`	`char`
	`unsigned char`

`INT32`	`long int`
`INT`	`int`
`REAL32`	`float`
`REAL64`	`double`

Table 11.3 Equivalent function return types

As an example, consider the C function `cfun` which returns `int`:

```
int cfun(int a);

#pragma IMS_nolink(cfun)

int cfun(int a)
{
⋮
}
```

This would be called from occam as an `INT FUNCTION` as follows:

```
#PRAGMA EXTERNAL "INT FUNCTION cfun (VAL INT x) = 20"
⋮
  cfun(42)
```

C function type `void`

A C function of type `void` must be called from occam as a `PROC`. For example:

```
void cfun(int a);

#pragma IMS_nolink(cfun)

void cfun(int a)
{
⋮
}
```

This can be called from occam in the following way:

```
#PRAGMA EXTERNAL "PROC cfun (VAL INT x) = 20"
⋮
  cfun(42)
```

Similarly, an occam `PROC` must be called from C as a `void` function.

Restrictions on functions that may be called

Because occam functions can only have `VAL` parameters, and these do not always have C equivalents, there are restrictions on the types of occam functions that can be called from C and vice-versa. For example, there are no equivalents of the occam `BOOL` type and so functions which require this type of parameter cannot easily be called.

Similarly, because C functions can only return a single value, only occam functions with a single return value can be called from C.

occam cannot call C functions which return structure types.

C functions that are called by occam must not modify any global variables, that is, they must be free from side-effects.

11.1.6 Global static base parameter

C uses an area of memory for static data. This requires a parameter to be passed to the called function to enable it to access the static area — this parameter is known as the Global Static Base or GSB. This parameter is added automatically by the compiler and is not normally visible to the programmer.

occam differs from C in that it does not use a static or heap area and so does not expect a GSB parameter to be passed to procedures. Similarly, occam programs do not pass a GSB pointer when procedures are called. In order to allow calls to work correctly between languages the presence of the GSB parameter must be taken into account.

There are two possible solutions to this problem:

1 A dummy GSB parameter can be provided in occam.

2 A compiler pragma can be used in the C program to specify that a function does not require a GSB parameter.

3 When calling occam from C, make use of the `call_without_gsb` function (see the *ANSI C Toolset Language and Libraries Reference Manual*).

The first two techniques can be used either on the routine being called or in the calling program, whichever is more appropriate.

In the examples below which show C functions called from occam, it is assumed that the C code does not use any static or heap memory. However, it will often be necessary for the occam calling program to allocate some memory for use by the C code as the static or heap area; a pointer to this memory is then passed as the first parameter when the function is called. This technique is described in more detail in section 11.1.8.

Method 1 — dummy GSB parameter.

A dummy parameter can be used either as a formal parameter for procedures which are to be called from C, or as an actual parameter for C functions which are being called from occam. For example the following occam function can be directly called from a C program:

```
INT FUNCTION ocfunc(VAL INT GSB, arg1, arg2)
  -- Note: dummy parameter GSB is not used
  INT return:
  VALOF
    ⋮
    RESULT return
:
```

Note: because the dummy parameter is not used, the occam compiler will generate a warning message but correct object code is still generated.

To call this version of `ocfunc` from a C program it is declared as an `extern` function (without the GSB parameter) and then called normally:

```
/* declare function as external */
extern int ocfunc(int arg1, int arg2);
⋮
/* call function */
ret = ocfunc(x, y);
```

The same method can be used to call a C function from occam by passing a dummy first parameter of type `INT`. For example the C function:

```
void cfun(int a)
{
⋮
}
```

Could be called from occam in the following way:

```
#PRAGMA EXTERNAL "PROC cfun (VAL INT GSB, x) = 20"
⋮
  VAL INT GSB IS 0:
  cfun(GSB, 42)
```

Method 2 — nolink pragma

In order to simplify mixing occam and C, the INMOS C compiler provides the `IMS_nolink` pragma which directs the specified function to be compiled without the static link parameter. Any *calls* of the function, within the scope of the pragma, will not have the GSB added to the parameter list. If the function is *defined* within the scope of the pragma then it will be compiled without the requirement for a static link parameter (the compiler will flag a serious error if the function requires access to static data).

As an example, consider the occam function `ocfunc` below:

```
INT FUNCTION ocfunc(VAL INT arg1, arg2)
  INT ret :
  VALOF
    ⋮
    RESULT ret
:
```

To call `ocfunc` from a C program it must first be declared as an `extern` function and then specified as not requiring the GSB parameter:

```
/* declare function as external */
extern int ocfunc(int arg1, int arg2);

/* specify that function has no GSB parameter */
#pragma IMS_nolink(ocfunc)

/* call function */
ret = ocfunc(x, y);
```

The same technique can be used to compile a C function which does not require a GSB parameter so that it can be called directly from occam. As an example, consider the C function below:

```
/* declare function before referencing */
void cfun(int a);

/* specify that function has no GSB parameter */
#pragma IMS_nolink(cfun)

/* define the function */
void cfun(int a)
{
⋮
}
```

This can be called from occam in the following way:

```
#PRAGMA EXTERNAL "PROC cfun (VAL INT x) = 20"
⋮
  cfun(42)
```

Method 3 — using `call_without_gsb` function

This method is applicable only when dynamically loading code using the ANSI C Toolset. It is described in the *ANSI C Toolset User Guide*.

11.1.7 Linking the program

After all the component parts of the program have been compiled, they must be linked together with any libraries required. The libraries that are required will depend on a number of factors such as the language that the main (calling) program is written in, whether the program communicates with the host, which library routines are used by the different language modules. Some guidelines for various configurations are given below.

Calling occam from C

When calling occam code from a C program, then the following library files must be linked with the compiled occam and C code.

- The C runtime library

  If the program uses the host file server then the *full* runtime library must be used. This can be linked in by using the linker indirect file `cstartup.lnk`.

  If the program does not use the host file server then the *reduced* runtime library must be used. This can be linked in by using the linker indirect file `cstartrd.lnk`.

- The standard occam compiler libraries will be required by most occam code. These libraries can be linked in by using the appropriate `occam`*x*`.lnk` linker indirect file.

- Any other C or occam modules or libraries referenced by the program must also be linked in.

Calling C from occam

When calling C code from an occam program, then the following library files must be linked with the compiled C and occam code.

- The standard occam compiler libraries can be linked in by using the appropriate `occam`*x*`.lnk` linker indirect file.

- If the main program is written in occam and allocates static or heap memory for C functions using the library procedures described in section 11.1.8, then the library `callc.lib` must be linked in.

- Any other C or occam libraries used must also be linked in.

- The *reduced* C library must be used as the called functions cannot make any host file server requests. The reduced runtime library can be linked in by using the `clibsrd.lnk` linker control file.

11.1.8 Allocating memory for C functions called from occam

The C runtime environment automatically provides C programs with a static area (for holding static data and external variables) and a heap area (for memory allocation). However occam does not provide these and so this memory must be explicitly allocated by the calling program before C functions are called. Four routines in the occam library `callc.lib` are used to set up and terminate C static and heap areas from occam for C functions that require them.

The static area

C static data is stored in a reserved area of memory called the static area which must be set up by the system and initialized. Each C function which uses static data needs to be able to find this area. In order to do this, every C function is passed, as the first parameter, a pointer to the start of the static area, the global static base (GSB). The static area must be set up and the GSB parameter passed explicitly by the calling occam code. This means that a call to a C function from occam will have one extra parameter compared to an equivalent call from C.

The heap area

The heap area is that area of memory from which the C memory allocation functions reserve their memory space. It is separate from the static area and requires a static area to be previously allocated because information about the heap is held in static variables.

The heap need not be set up if it is not required, but remember that it may be used implicitly by a library call.

Providing static and heap

Some simple C functions may not require static or heap areas and may be called more easily without using the special library routines. When calling a C function therefore, the first step is to decide whether static and heap areas are required.

Deciding whether a static area is required

For many C functions it may not be immediately obvious whether static or heap is required (the heap area requires a previously set-up static area). For example, some, but not all, library functions require static and heap areas and so, because it would be difficult to distinguish those that do, a static and heap area should be assumed whenever a library function is called.

Because of the difficulty in covering all types of functions, the following series of rules is offered as a way of determining whether a function requires static or heap.

The rules include the most common reasons for a C function requiring static or heap memory.

- If the function uses static variables then static is required.
- If the function accesses external variables then static is required.
- If the function includes an automatic structure or union initializer then static is required.
- If the function uses any functions from the runtime library then static and heap may be required.

Functions which fail all the above tests will probably not require static or heap, and can be called without using any of the static or heap library functions.

Calling functions which do not require static or heap

C functions which do not require static or heap can be called as described in section 11.1.6.

Calling functions which do require static or heap

For C functions which require static and/or heap the space must be set up in the occam code before the function is called, and terminated when no longer required. These operations are performed by procedures supplied in the library `callc.lib`. This library is supplied as part of the ANSI C toolset — do *not* use any previous version of the library which was supplied as part of an occam toolset.

The library `callc.lib` provides four occam procedures for initializing static and heap areas and terminating them after use. The routines are summarized in table 11.4 and described in more detail below.

Procedure	Description
`init.static`	Initializes an area of memory for use as the static area.
`init.heap`	Initializes an area for use as the heap area.
`terminate.heap.use`	Terminates heap usage.
`terminate.static.use`	Terminates static usage.

Table 11.4 Library procedures to support memory allocation

```
PROC init.static([]INT static.area, INT required.size, GSB)
```

`init.static` is used to set aside and initialize an area of memory for use as a C static area before any C functions are called. The static area is declared as an integer array in the calling occam program.

Two integer values are returned in the procedure parameters:

`required.size`	The number of words of static space required.
`GSB`	A pointer to the base of the array which will act as the global static base.

Note: the size of the integer array is equivalent to the number of words of static space required. One element of the integer array is equivalent to one word of memory. If an error occurs on initializing the static area the value `MOSTPOS INT` is returned instead of the required size.

The procedure can be used to check the size of static area required by checking the value returned in the second parameter. For example:

```
#USE "callc.lib"

INT required.size, GSB:
[STATIC.SIZE]INT static.area:

SEQ
  init.static(static.area, required.size, GSB)
  IF
    required.size > STATIC.SIZE
      ...  not enough space reserved
    TRUE
      ...  array is big enough
```

Another possible way of using `init.static` is to reserve a large amount of memory for use by the C function. To do this an initial call to `init.static` would be made with an array size of zero to obtain the required size, followed by a second call which would set up a segment of memory as the static area. The rest of the memory could be used by the occam program for its own purposes, perhaps to allocate the C heap. For example:

```
#USE "callc.lib"

INT required, GSB:
[VERY.BIG.NUMBER]INT memory :

SEQ
  -- check the static requirement
  init.static([memory FROM 0 FOR 0], required, GSB)

  -- allocate required amount of memory for static
  static.area IS [memory FROM 0 FOR required]:
  -- rest is available for other purposes
  memory.left IS [memory FROM required FOR
                  (VERY.BIG.NUMBER - required)]:
  SEQ
    -- now use allocated memory as static
    init.static(static.area, required, GSB)
    ...  rest of program
```

`PROC init.heap(VAL INT GSB, []INT heap.area)`

`init.heap` is used to set aside an area of memory for use as a C heap before any C functions are called. The first argument is the GSB pointer returned by `init.static`, which is required because the memory allocation routines make use of static data.

Like the static area, the heap area is declared as an integer array. This array must be large enough to accommodate all calls to the C memory allocation functions. The size of the integer array is equivalent to the number of words of heap area required. One element of the integer array is equivalent to one word of memory.

If the heap is used by a function before `init.heap` has been called the C memory allocation functions will fail with their normal error returns.

`PROC terminate.heap.use(VAL INT GSB)`

`terminate.heap.use` should be called when the heap is no longer required, i.e. when no more C functions will be called. It provides a clean way of terminating the use of the heap.

Once the heap terminate procedure has been called, the state of the heap is undefined.

`terminate.heap.use` must be called *before* terminating the static area because the heap is accessed using static variables.

```
PROC terminate.static.use(VAL INT GSB)
```

`terminate.static.use` should be called when the static area is no longer required, i.e. when no further calls to C will be made. It provides a clean way of ending the use of the C static area.

Once the static terminate procedure has been called, the state of the static area is undefined.

Example

The following example illustrates how these library procedures can be used to set up and terminate the static and heap areas for a C function. The C function to be called is:

```
#include <stdlib.h>

int c_func(int n, int release){

  static int *ptr = NULL;
  int i;

  if (ptr == NULL){
    ptr = (int *) malloc(n);

    if (ptr == NULL)
      return 1;
  }

  for (i = 0; i < n / sizeof(int); i++)
      ptr[i] = i;

  if (release){
    free (ptr);
    ptr = NULL;
  }

  return 0;
}
```

The occam code to call this function (on a 32 bit transputer) is shown below:

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"
#USE "callc.lib"  -- the 'calling C' functions.

#PRAGMA TRANSLATE C "c_func"

-- declare the C function as an occam descriptor.
#PRAGMA EXTERNAL "INT FUNCTION C(VAL INT GSB,x,free) = 200"

PROC mixed (CHAN OF SP fs, ts, []INT freemem)
  INT GSB, required.size :

  -- Allow very large static and heap area sizes
  VAL static.size IS 4000 :
  VAL heap.size   IS 4000 :
  [static.size]INT static.area :
  [heap.size]INT   heap.area :

  SEQ
    -- set up static.area as the static area
    init.static(static.area, required.size, GSB)
    -- now check for error
    IF
      required.size > static.size
        so.write.string(fs, ts,
                 "error initialising static*n")
      TRUE
        INT fail:
        SEQ
          -- Set up the heap area.
          -- Note that GSB is the first parameter
          init.heap(GSB, heap.area)

          -- Call the C function. Note that the GSB
          -- is passed as the first parameter.
          fail := C (GSB, 20000, 0)
          IF
            fail = 0
              so.write.string(fs, ts, "malloc OK*n")
            TRUE
              so.write.string(fs, ts, "malloc failed*n")
    -- now tidy up the stack and heap allocated
    terminate.heap.use(GSB)
    terminate.static.use(GSB)
    -- and exit
    so.exit(fs, ts, sps.success)
:
```

The occam program must be compiled and then linked with the compiled C function, the memory allocation library, the reduced C runtime library, the occam host i/o library, and the standard occam libraries. In this example (assuming that the C source code is in a file called `cfunc.c` and the occam source is in a file called `mixed.occ`) the set of files to be linked is:

`mixed.tco`	compiled occam program
`cfunc.tco`	compiled C function
`clibsrd.lnk`	linker indirect file for the C reduced runtime library
`hostio.lib`	occam i/o library
`callc.lib`	call C library
`occama.lnk`	linker indirect file listing standard occam libraries for code compiled for transputer class TA

Sources can be found on the toolset examples directory. Standard libraries and linker indirect files are available on the toolset libraries directory.

The linker allows files to either be specified on the command line or listed in an indirect file. Because there are several files required in this instance, it may be easier to supply a linker indirect file. This file can also include a `#mainentry` directive to define the entry point of the program, in this case the top level occam procedure "`mixed`". To do this create a text file called `callc.lnk`, containing the following lines:

```
mixed.tco
cfunc.tco
#include clibsrd.lnk
hostio.lib
callc.lib
#include occama.lnk
#mainentry mixed
```

The correct linker command line (using the default processor T414 in HALT mode) would be as follows:

```
ilink -f callc.lnk                                    (UNIX)

ilink /f callc.lnk                                    (MS-DOS/VMS)
```

Details of the operation of the linker can be found in chapter 9 in the *Toolset Reference Manual.*

Once linked, the program can be collected and run in the usual way. The output of the program is the message 'malloc OK'.

11.1.9 Restrictions and caveats

General

A number of restrictions must be observed when calling routines written in one language from a program in a different language:

1 The formal and actual parameters (and function return types) must be compatible. See sections 11.1.3 and 11.1.5 for more detail.

2 As occam does not have 'external' variables, there can be no common data between the calling program and the called routine. Therefore, the only way that data can be transferred between them is by means of parameters (and return values). The called procedure may also use channels to communicate with other parts of the program that are running in parallel.

3 No function or procedure which requires direct communication with the *host file server* may be called.

Rules for importing C code

The following restrictions apply to C functions which are to be called from an occam program:

1 Stack checking should not be enabled in any C function to be called from occam.

2 Only C functions linked with the reduced C runtime library, can be called from occam, i.e. those which do not require any server communication.

3 Imported C functions which return a single value (other than a pointer) must not have any 'side-effects'. They must not: alter parameters and variables (except those declared within the function); perform channel or host i/o; call functions which do have side-effects; perform parallel operations; use timer delays; or perform heap operations.

4 The following functions cannot be called in the imported C code:

```
clock()

exit()

exit_terminate()

exit_noterminate()

exit_repeat()

get_detail_of_free_stack_space()
```

Rules for importing occam code

There are certain rules which govern the calling of occam code from C:

1 occam functions that return more than a single value may *not* be called.

2 The occam procedure or function to be called must be at the *outer level* of a compiled module.

3 `INLINE` procedures and functions cannot be called from C.

4 The occam code must not use vector space, or call any other occam code which uses vector space. Arrays, if used, should be explicitly placed within workspace or the code should be compiled with the `V` option to disable the use of separate vector space.

  Some occam libraries supplied with the occam 2 toolset use vector space and therefore cannot be called from C. These are:

```
hostio.lib  streamio.lib  msdos.lib
```

5 There must be enough workspace for the called procedures or functions on the stack of the calling program. It is the programmer's responsibility to ensure that this is the case.

6 There must be no *aliasing* between the parameters to occam functions or procedures and the destination of the result. In other words the same variable must not be used as both a parameter which will be read, and as a result. The occam compiler checks that this is so for occam procedures and functions called within an occam program.

  The presence or absence of alias checking when the occam code is compiled has no effect on this rule.

  As an example consider the occam function:

```
INT FUNCTION succ (VAL INT n) IS n + 1 :
```

  If this is called from within an occam program, the compiler will check to see whether the parameter and result are aliased; if they are then the compiler will generate temporary variables as necessary. So, for example, the occam call `i := succ(i)` may be compiled with a temporary variable for the function result, which is then copied to the variable `i`. The C compiler is not able to perform these checks and so, if this function is called from C, it is up to the programmer to ensure that there is no aliasing. A suitable calling sequence could be:

```
int tmp;

tmp = succ(i);
i = tmp;
```

  Note that there may be mutual aliasing between `VAL` parameters as these are only read, not written.

11.2 occam interface procedures

The following sections describe a set of interfaces provided to allow complete programs written in C to be called from occam. This might be done for various reasons, for example to allow a C program to be used with the occam configurer `occonf`, or to provide some simple modification of the runtime environment of the program — e.g. initializing some external hardware before the application code starts, or intercepting the program's communications with the host file server.

By specifying the appropriate entry point for a C program, it is given an occam-like procedural interface allowing the program to be called from an occam program. The code produced in this way is known as an *occam equivalent process* as it makes the program look like an occam process with channels for input and output.

11.2.1 Interface code

The occam interface code described here provides a number of fixed interfaces to a C program. There are three types of interface code, known as types 1, 2, and 3. Descriptions and process diagrams for the three interfaces are given below.

Type 1

This interface is used when the C program runs on a single transputer and communicates only with the host file server. This interface is used with the full version of the C runtime library.

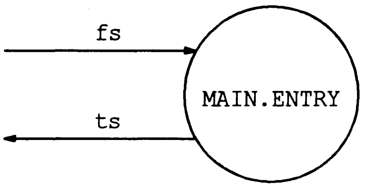


Figure 11.1 Type 1 interface

Type 2

This interface is used when the C program communicates with other processes as well as the host file server. This interface is used with the full version of the C runtime library.

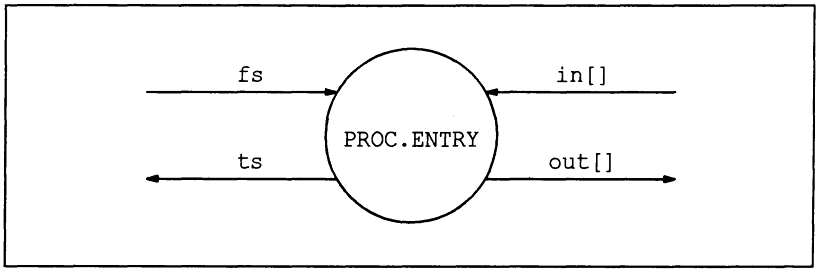


Figure 11.2 Type 2 interface

Type 3

This interface is similar to the type 2 interface except that there is no access to the host file server. The interface must be used with the *reduced* version of the C runtime library, which does not contain any functions which require access to `iserver` facilities such as the host file system.

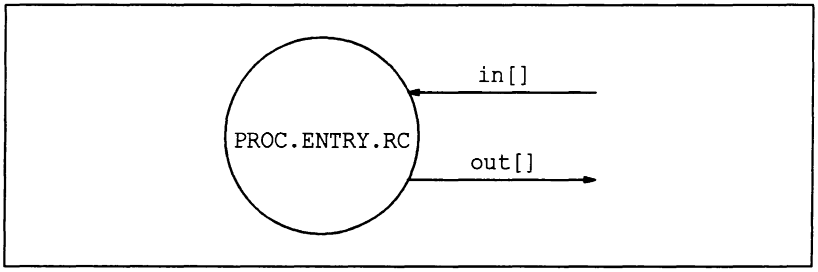


Figure 11.3 Type 3 interface

Channel arrays

The Type 2 and type 3 interfaces have arrays of channels which enable the C program to communicate with other processes in the program. These arrays are mapped directly onto the channel arrays which form part of the standard parameter list of the C `main` function (see section 11.2.7).

These channel arrays actually appear as arrays of integers in the occam parameter lists — this allows *pointers* to channels to be passed to the C program which provides a more flexible way of mapping channels onto the arrays. Because occam does not support pointers directly, two library procedures are provided to assign channel pointers to array elements (for more information on these, see the examples below and the *occam 2 Toolset Language and Libraries Reference Manual*).

Reserved channels

Two of the input channels and two of the output channels in the Type 2 and Type 3 occam interface procedures (i.e. `in[0]`, `in[1]`, `out[0]` and `out[1]`) are reserved. No program should use these channels. They are reserved as follows:

`out[0]`	Reserved for diagnostic output.
`in[0]`	Reserved for diagnostic input.
`out[1]`	Messages from the runtime library to the host file server.
`in[1]`	Responses from the host file server to the runtime library.

11.2.2 Parameters to the C program

Parameters to the C `main` function are described by the following function prototype:

```
#include <channel.h>

int main (int argc, char *argv[], char *envp[],
          Channel *in[], int inlen,
          Channel *out[], int outlen);
```

Where:

- `argc` — the number of arguments passed to the program from the command line, including the program name.
- `argv` — an array of pointers to those arguments.

  Note: for programs linked with the reduced runtime library (i.e. using the Type 3 interface), `argc` is set to 1 and the first element of `argv` is a pointer to an empty string.
- `envp` — included for compatibility with previous toolsets — in this implementation, this parameter is always set to `NULL`.
- `in` — an array of input channels.
- `inlen` — the size of the array `in`.
- `out` — an array of output channels.
- `outlen` — the size of the array `out`.

The channel arrays `in` and `out` in the C program are passed from the interface procedures, and can be set up as described below. Where applicable, these channels can be used by the C code to communicate via channels passed in from the calling occam program. Note, however, that the first two elements in the arrays are reserved for use by the C program's runtime system and cannot be used by the application program.

11.2.3 Stack and heap requirements

Data storage (workspace) requirements for C programs are provided by arrays in the occam code. Stack, static and heap requirements vary from program to program. The workspace arrays passed to the program must be large enough to accommodate:

- the stack needed by the program when it runs
- all the static data required by the program
- the heap used by the program and the runtime libraries.

Stack overflow may lead to unpredictable behavior by the program. For this reason it is best to run a program initially with a large combined stack and heap. Later, after the program has been run to determine stack and heap usage, it can be modified to use a separate stack and heap of the appropriate sizes. The use of a separate array for the stack allows the stack to be placed in the transputer's internal memory to optimize the performance of the program. Methods for optimizing memory usage are described in: *Performance Improvement with the INMOS Dx305 occam 2 Toolset* (supplied with the toolset); and INMOS Technical Note 55 *Using the occam toolsets with non-occam applications.*

A minimum stack size of 512 words is recommended.

Stack overflow detection

Failure or unpredictable behavior of programs may be due to stack overflow. To obtain an estimate of the amount of stack used by a program:

1 Build all C code with stack checking enabled.

2 Call the function **`max_stack_usage`** at the end of the program, this will return an approximation of the amount of stack used by the program.

A test for stack overflow in a program is to use the procedure outlined below:

1 Initialize the bottom few words of the stack (a falling stack is used) to some easily recognizable 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 initialized have been changed then stack overflow is likely.

3 Increase the stack size and try again.

A similar method can be used to determine static data and heap requirements, except that these are allocated upwards in memory. The following occam fragment gives an example of initializing the bottom of the stack:

```
SEQ i = 0 FOR SIZE ws1
  ws1[i] := #DEFACED
```

Stack overflow in the C parts of the program can also be detected by using the stack checking mechanism built into the C compiler and libraries.

11.2.4 Type 1 interface definition

The Type 1 interface is used when the C program does not communicate with any other process apart from the host file server.

The parameters for the Type 1 procedure are: a pair of channels to communicate with the host file server; and two arrays to provide the C program's heap, static and stack space.

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)
```

The parameters to this procedure are:

- `fs` — a channel from the host file server to the C program.
- `ts` — a channel from the C program to the host file server.

  The channels `fs` and `ts` are connected to the channels `in[1]` and `out[1]` which are passed as parameters to the C program — these are provided for the use of the C runtime libraries only, and should not be used by the application code.
- `free.memory` — used by the C program for its heap and static areas.

  This array is generally used to pass the free memory which is available to the C program after the all the code has been loaded.
- `stack.memory` — used by the C program for its runtime stack (if the size of the array is non-zero).

  If the size of the `stack.memory` array is zero then the `free.memory` array is used for the program's runtime stack as well as for the static and heap data areas.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to 2
- `in[0]` and `out[0]` are set to NULL

- `in[1]` is a pointer to the `fs` channel and is used by the C runtime system to communicate with the host
- `out[1]` is a pointer to the `ts` channel and is used by the C runtime system to communicate with the host

Example

The following example is an occam procedure, `call.prog1`, which calls a C program via the `MAIN.ENTRY` procedure interface:

```
#INCLUDE "hostio.inc"

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

  #USE "centry.lib"           -- C interface code

  [100000]INT heap :          -- static and heap space
  [1024]INT   stack :         -- stack for program
  PLACE stack IN WORKSPACE :  -- Put on chip

  -- call program
  MAIN.ENTRY(fs, ts, heap, stack)
:
```

11.2.5 Type 2 interface definition

The Type 2 interface is used when building a program that will communicate with other processes as well as with the host file server.

The parameters for the Type 2 procedure are: a pair of channels to communicate with the host file server; a *flag* value to control the use of memory by the C program; two arrays to provide the C program's heap, static and stack space; and a pair of channels for passing channel pointers to the C program.

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)
```

The parameters are described below:

- `fs` — a channel from the host file server to the C program.
- `ts` — a channel from the program to the host file server.

  The channels `fs` and `ts` are connected to the channels `in[1]` and `out[1]` which are passed as parameters to the C program — these are provided for the use of the C runtime libraries only, and should not be used by the application code.

- `flag` — indicates whether one or two workspaces are to be used.

  If the value of `flag` is set to 0 then the program will run with two workspace areas; one for static and heap data, the other for the runtime stack. If the value of `flag` is set to 1 then the program will run with a single combined workspace.

- `ws1` — used by the C program for its workspace.

  If `flag` is 0 then this array is used only for the runtime stack, if `flag` is 1 then it is used as the program's combined workspace (static, heap *and* stack).

- `ws2` — used by the C program as its static/heap workspace when `flag` is set to zero, otherwise unused.

- `in` — an array of pointers to occam channels going to the C program.

- `out` — an array of pointers to occam channels going from the C program.

Note: The first two elements in the channel pointer arrays `in` and `out` are reserved for use by the C program's runtime system and cannot be used by the program.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to the number of elements in the occam arrays `in` and `out`

- `in[0]` and `out[0]` are set to NULL

- `in[1]` is a pointer to the `fs` channel and is used by the C runtime system to communicate with the host

- `out[1]` is a pointer to the `ts` channel and is used by the C runtime system to communicate with the host

- The remaining elements of the arrays `in` and `out` are set to the values in the corresponding elements of the occam arrays

Example

The following example is an occam procedure, `call.prog2`, which calls a C program via the `PROC.ENTRY` procedure interface:

```
#INCLUDE "hostio.inc"

PROC call.prog2 (CHAN OF SP fs, ts,
                 CHAN OF COMM to.process,
                 CHAN OF COMM from.process)

  #USE "hostio.lib"
  #USE "centry.lib"       -- C interface code

  VAL flag IS 1 :         -- combined heap and stack
  [100000]INT ws1 :       -- stack and heap for program
  [1]INT ws2 :            -- dummy workspace for program
  [3]INT in, out :        -- channel pointers (not used)

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

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

    -- call program
    PROC.ENTRY(fs, ts, flag, ws1, ws2, in, out)
    so.exit(fs, ts, sps.success)
:
```

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.)

11.2.6 Type 3 interface definition

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

Programs using the Type 3 interface *must* be linked with the reduced C runtime library.

The parameters for the Type 3 procedure are: a *flag* value to control the use of memory by the C program; two arrays to provide the C program's heap, static and stack space; and a pair of channels for passing channel pointers to the C program.

Procedural interface

The interface for Type 3 equivalent occam processes is defined below:

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

The parameters are described in the following list.

- `flag` — indicates whether one or two workspaces are to be used.

  If the value of `flag` is set to 0 then the program will run with two workspace areas; one for static and heap data, the other for the runtime stack. If the value of `flag` is set to 1 then the program will run with a single combined workspace.

- `ws1` — used by the C program for its workspace.

  If `flag` is 0 then this array is used only for the runtime stack, if `flag` is 1 then it is used as the program's combined workspace (static, heap *and* stack).

- `ws2` — used by the C program as its static/heap workspace when `flag` is set to zero, otherwise unused.

- `in` — an array of pointers to occam channels going to the process.

- `out` — an array of pointers to occam channels coming from the process.

Note: The first two elements in the channel pointer arrays `in` and `out` are reserved for use by the C program's runtime system and cannot be used by the occam program.

Parameters to C program

The channel array parameters to the C `main` function are set up as follows:

- `inlen` and `outlen` are set to the number of elements in the occam arrays `in` and `out`

- `in[0]`, `in[1]`, `out[0]` and `out[1]` are are set to NULL

- The remaining elements of the arrays `in` and `out` are set to the values in the corresponding elements of the occam arrays

Example

The following shows how to call a Type 3 equivalent occam process from occam source, and how to set up the parameters required. The example consists of an occam procedure 'call.prog3' within which a C program is called.

```
PROC call.prog3 (CHAN OF COMM to.process,
                 CHAN OF COMM from.process)

  #USE "centry.lib"        -- C entry point library

  VAL flag IS 0 :          -- separate heap and stack

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

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

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

    -- call program
    PROC.ENTRY.RC(flag, ws1, ws2, in, out)
:
```

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

The first statement sets up a pointer to the output channel, using the procedure `LOAD.OUTPUT.CHANNEL`. The second statement sets up a pointer to the input channel, using the procedure `LOAD.INPUT.CHANNEL`. Note that the first two input and output channels are reserved by the runtime system even though there is no host communication taking place.

11.2.7 Building the occam equivalent process

The occam equivalent processes built from these interfaces can be called from an occam program in the same way as any other occam procedure. Note that, because the interface procedures have fixed names, there can only be one process of a particular type in each linked unit. However, multiple C programs called in this way may be placed on a processor by the configurer.

Once all the component C and occam code for the complete program has been compiled, it is linked with the C runtime libraries, the occam entry points library and any other occam libraries required. The program is then configured and a bootable code file produced.

The occam interface code is supplied in the library `centry.lib`. The C libraries can be linked by using the linker control file `clibs.lnk`, for the *full* runtime library, or `clibsrd.lnk`, for the *reduced* runtime library. For example, consider a program that consists of the following compiled files:

- `main.tco` — the compiled C program to be called from occam
- `wrap.tco` — the compiled occam code that calls the interface procedure

This program can be linked with the full run-time libraries, for a 32 bit transputer, using the following command:

```
ilink wrap.tco main.tco callc.lib -f clibs.lnk -f occama.lnk
```
(UNIX toolsets)

```
ilink wrap.tco main.tco callc.lib /f clibs.lnk /f occama.lnk
```
(MS-DOS/VMS toolsets)

12 EPROM programming

INMOS EPROM software is designed so that programs can be developed, booted onto a network via link and tested using the INMOS toolset. Once they are working, they can be placed in ROM with only minor change.

12.1 Introduction

During development, software is booted onto a network from a link connecting the network to the host computer. Then the software is prepared for a ROM, which is attached to the root transputer in the network.

Figure 12.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

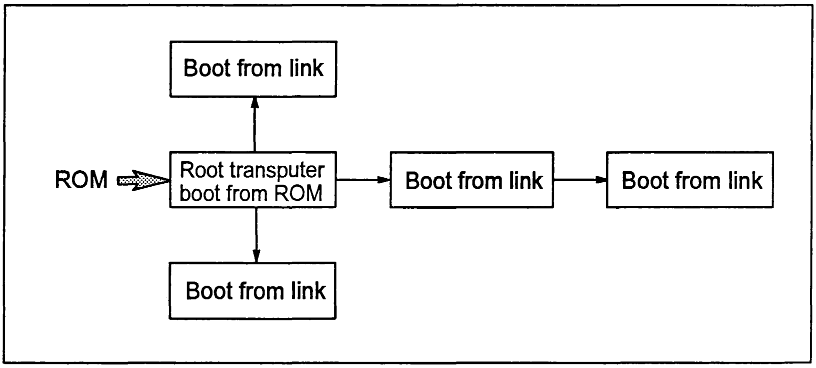


Figure 12.1 Loading a network from ROM

To prepare software to be booted from ROM, rather than to be booted from link, the following two steps must be taken:

1. Give different options to the configurer and collector tools so that they produce ROM-bootable code.
2. Run the `ieprom` tool to produce a file or set of files suitable for blowing into EPROM.

Figures 12.2 and 12.3 illustrate the stages of preparing ROM-bootable software.

Figure 12.2 shows an occam program compiled and linked for a single processor. Figure 12.3 shows a configured program, consisting of one or more linked units, connected together and allocated to processors as described in a configuration file.

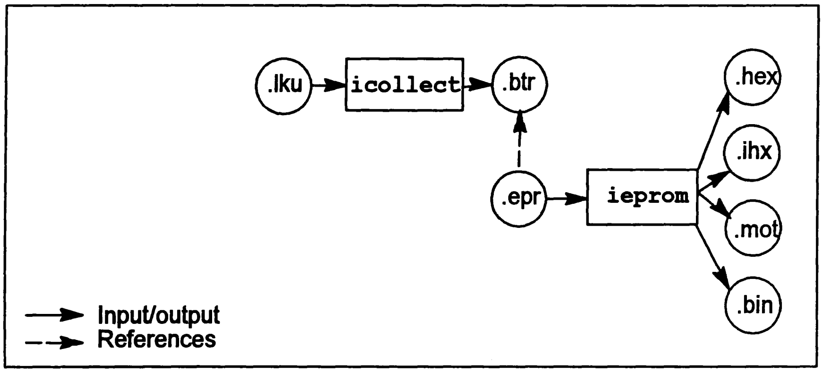


Figure 12.2 Preparation of ROM-bootable software (single occam program)

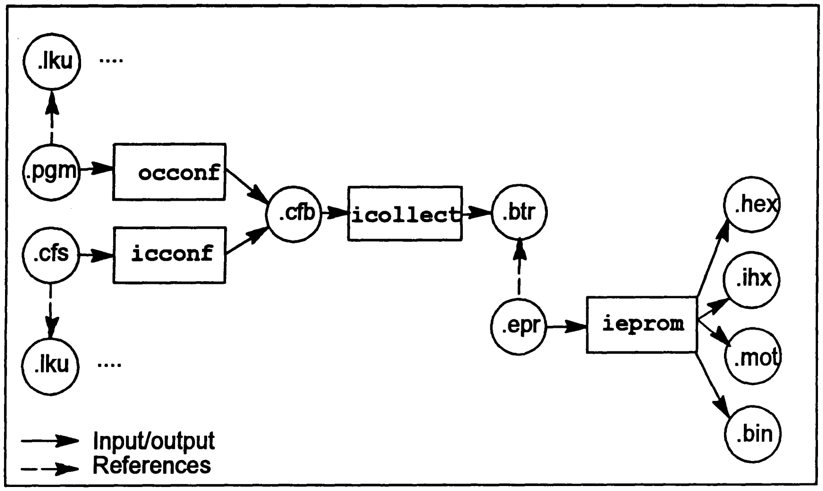


Figure 12.3 Preparation of ROM-bootable software (configured program)

12.2 Processing configurations

The processing configuration used will depend on the number of software processes, the number of transputers available to run the code and whether the code is to run from ROM or RAM. The following sections outline the possible configurations.

When preparing FORTRAN or C code to be booted from ROM the configurer must be used in order to specify the size of stack and heap. This applies even when the application consists of a single process running on a single processor. A single occam process can be configured or prepared as a single, linked program.

12.2.1 Single processor, run from ROM

The application process is prepared as one or more processes, connected as described in a configuration file. If the application consists of a single occam program then it can be prepared without using the configurer. It is then run on a single processor, with the code in ROM, and the RAM is used as the data area.

12.2.2 Single processor, run from RAM

The application process is prepared as one or more processes, connected as described in a configuration file. If the application consists of a single occam program then it can be compiled and linked without using the configurer. When booted from ROM, the processor copies the code into RAM and runs it, using the RAM for the data area.

12.2.3 Multiple process, multiple processor, run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads its own code into RAM, and loads the rest of the network via its links. Each processor then sets off its own processes, and the application runs. (This is the configuration shown in figure 12.1).

12.2.4 Multiple process, multiple processor, root run from ROM, rest of network run from RAM

The application is prepared as a collection of processes, connected and allocated to processors as described in a configuration file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor loads the rest of the network via its links, and then continues to run its own code from the ROM.

12.3 The EPROM tool: `ieprom`

The EPROM tool `ieprom` takes the output of the collector, and produces a file or set of files suitable for blowing into an EPROM. The following output formats are supported:

- Binary
- Hex
- Intel hex format
- Intel extended hex format
- Motorola S-record format

ieprom supports the production of code files in *block mode*, which allows the code to be placed in a set of different files. This is useful to program EPROMS organized as separate byte-wide devices, or where the EPROM programming device does not have enough memory to hold the entire image.

ieprom also supports the inclusion in the EPROM image of a *memory configuration.* Some 32-bit transputers have a configurable memory interface which can be initialized from a fixed area in the ROM, when the transputer is reset. A particular memory configuration can be specified to **ieprom** in a text file. These files are known as memory configuration files and normally have the file extension **.mem**. The format of these files, and the facility to edit them using an interactive tool called **iemit** is described in chapter 6 of the accompanying *Toolset Reference Manual.*

ieprom is driven by a control file which normally has the file extension **.epr**. A detailed description of **ieprom** and its control file is given in chapter 7 of the accompanying *Toolset Reference Manual.*

12.4 Using the configurer and collector to produce ROM-bootable code

To produce code suitable for running in ROM or RAM, the configurer and collector tools must be specified with the appropriate command line options. The following options are used to configurer single and multi-processor programs and to collect unconfigured single processor programs:

- The **ro** option specifies that the code is to run in ROM.
- The **ra** option specifies that the code is to run in RAM.
- The **rs** option specifies the ROM size (if not specified in configuration file). This option does not apply to the occam configurer **occonf**, see below.

In addition, if using **icconf** (the C configurer), the **P** option must be used in order to specify the root processor name.

If using **occonf**, the **NETWORK** description in the configuration file should indicate:

- which processor is the root processor, by setting its **root** attribute to **TRUE**
- the size of the ROM on that processor, by setting its **romsize** attribute to the appropriate value, in bytes.

The collector will add the appropriate ROM bootstrap to the application code and the output file will be given the extension **.btr**.

12.5 Summary of EPROM tool steps for different configurations

12.5.1 Using `icconf`

	Compile and link	**Configure**	**Collect**	**EPROM**
Single processor, run from ROM.	Compile and link a set of units, one per process.	Configure with the `ro`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the `ra`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the `ra`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor root runs from ROM, rest of network runs from RAM.	Compile and link a set of units, one per process.	Configure with the `ro`, `rs` and `p` options.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

12.5.2 Single processor unconfigured occam program

	Compile and link	**Configure**	**Collect**	**EPROM**
Run from ROM.	Compile and link program.	Not needed.	Collect with the `ro` and `t` options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Run from RAM.	Compile and link program.	Not needed.	Collect with the `ra` and `t` options.	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

12.5.3 Using `occonf`

	Compile and link	**Configure**	**Collect**	**EPROM**
Single processor, run from ROM.	Compile and link a set of units.	Configure with the `ro` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Single processor, run from RAM.	Compile and link as a set of units.	Configure with the `ra` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, run from RAM.	Compile and link a set of units.	Configure with the `ra` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.
Multiple processor, root runs from ROM, rest of network runs from RAM.	Compile and link a set of units.	Configure with the `ro` option.	Collect	Run EPROM tool to add memory interface (if necessary), and produce EPROM files.

13 Low level programming

This chapter describes a number of features of the toolset occam 2 compiler which support low-level programming of transputers. These are as follows:

Allocation This allows a channel, a variable, an array or a port to be placed at an absolute location in memory.

RETYPING channels and creating channel array constructors. These facilities enable channels to be manipulated.

Code insertion This allows sections of transputer machine code to be inserted into occam programs.

Dynamic code loading A set of library procedures is provided that allows an occam program to read in a section of compiled code (from a file, for example) and execute it.

Extraordinary use of links A set of library procedures is provided which allow link communications which have not completed to be handled by timeout, or be aborted by another part of the program.

Scheduling Using the predefined routine **RESCHEDULE** to reschedule processes.

Setting the error flag The transputer error flag can be explicitly set using the predefined routine **CAUSEERROR**.

13.1 Allocation

Allocation is performed using the occam **PLACE** statement, which is defined formally as follows:

allocation = **PLACE** *name* **AT** *expression* :

(Section A.3.2 of the *occam 2 Toolset Language and Libraries Reference Manual* provides details of the **PLACE** statement).

The **PLACE** statement allows a variable to be assigned to a specific memory location. The variable can be a scalar variable, array variable, channel, or port. 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 must be inserted immediately following the declaration of the variable to which it refers e.g.

```
int x, y, z :
PLACE x .....
PLACE y .....          is correct

int x :
int y :
PLACE x .....          is incorrect
```

13.1.1 The PLACE statement

Normally the PLACE statement should not be used to force critical arrays or variables into on-chip RAM. The occam compiler allocates memory according to the scheme outlined in Appendix B of the *occam 2 Toolset Language and Libraries Reference Manual*, 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 5.6.

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

```
PLACE chan AT 1:
```

Addresses are calculated in this way so that the transputer links can be accessed using code that is independent of the word length. The links are mapped to addresses 0, 1, 2...7. (See section 13.1.3).

Translation from a machine address to the equivalent occam address PLACE value can be achieved by the following declaration:

```
VAL occam.addr IS
      (machine.addr><(MOSTNEG INT)) >> w.adjust:
```

where: w.adjust is 1 for a 16-bit transputer and 2 for a 32-bit transputer.

All placed objects must be word aligned. If it is necessary to access a BYTE object on an arbitrary boundary, or an INT16 object on an arbitrary 16-bit 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 32-bit transputer use the following:

```
[4]PORT OF BYTE io.regs.vec :
PLACE io.regs.vec AT #30000000 :
io.register IS io.regs.vec[1] :
```

The `PLACE` statement must be placed immediately after the declaration of the variable.

Placement may be used on transputer boards to access board control functions mapped into the transputer's address space. For example, on a TRAM with a subsystem, 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 a TRAM:

```
PROC reset.tram.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.

A more specific example of how to reset B004 type boards is given in the examples directory `examples/manuals/assert`.

13.1.2 Allocating specific workspace locations

A number of specialized transputer instructions require specific workspace placings. For example, the instructions `POSTNORMSN`, `OUTBYTE`, `OUTWORD` and the disabling `ALT` instructions all use workspace location 0. To accommodate this the compiler supports the following allocation:

```
PLACE name AT WORKSPACE n:
```

where: `n` is a constant integer. (See Appendix A in the *occam 2 Toolset Language and Libraries Reference Manual* for syntax details).

This is used to ensure that a variable is allocated a particular position within a procedure or function's workspace. The compiler ensures that at least `n` words of workspace are allocated, and that no other variables are placed at that address. The compiler will warn if a variable **PLACED AT WORKSPACE** `n` is in scope when its own workspace allocation requires to use that workspace location, or when another is **PLACED** at the same location.

For example on a T425, the **POSTNORMSN** instruction can be used to pack a floating point number; it requires an exponent to be previously stored at workspace offset 0. The following code may be used:

```
REAL32 FUNCTION pack (VAL INT guard, frac, exp, sign)
  REAL32 result :
  VALOF
    INT temp :
    PLACE temp AT WORKSPACE 0 :
    SEQ
      temp := exp
      ASM
        LDAB guard, frac
        NORM
        POSTNORMSN
        ROUNDSN
        LDL sign
        OR
        ST result
    RESULT result
:
```

(For the background on this example, see the *Transputer instruction set –a compiler writer's guide*, section 7.11.2). Use of the **ASM** construct is described in section 13.3.

13.1.3 Allocating channels to links

When mapping channels to specific transputer links, the channel word is placed at the specified address for scalar channels. Arrays of channels, however, are mapped as arrays of pointers to channels:

```
PLACE scalar.channel AT n:
```

places the channel word at that address.

```
PLACE array.of.channels AT n:
```

places the array of pointers at that address.

The following two code fragments illustrate the placement of channels on links.

```
CHAN OF ANY   in.link0, out.link0 :
PLACE   in.link0  AT link0.in:
PLACE   out.link0 AT link0.out:

CHAN OF ANY   in.link1, out.link1 :
PLACE   in.link1  AT link1.in:
PLACE   out.link1 AT link1.out:

CHAN OF ANY   in.link2, out.link2 :
PLACE   in.link2  AT link2.in:
PLACE   out.link2 AT link2.out:

CHAN OF ANY   in.link3, out.link3 :
PLACE   in.link3  AT link3.in:
PLACE   out.link3 AT link3.out:

CHAN OF ANY   in.event :
PLACE   in.event  AT event.in:
```

or:

```
CHAN OF ANY out.link0, out.link1, out.link2, out.link3 :
PLACE out.link0 AT link0.out :
PLACE out.link1 AT link1.out :
PLACE out.link2 AT link2.out :
PLACE out.link3 AT link3.out :
[4]CHAN OF ANY outlink IS [out.link0, out.link1,
                           out.link2, out.link3] :

CHAN OF ANY in.link0, in.link1, in.link2, in.link3 :
PLACE in.link0 AT link0.in :
PLACE in.link1 AT link1.in :
PLACE in.link2 AT link2.in :
PLACE in.link3 AT link3.in :
[4]CHAN OF ANY inlink IS [in.link0, in.link1, in.link2,
                          in.link3] :
```

Link addresses are defined in the include file `linkaddr.inc` that is supplied with the toolset.

Although shown here as `CHAN OF ANY` channels you should use specific occam channel protocols wherever possible to ensure that channels are properly checked at compile time.

13.2 RETYPING channels and creating channel array constructors

Channels may be `RETYPE`d, (see also section A.2.1. of the *occam 2 Toolset Language and Libraries Reference Manual*). This allows the user to change the protocol on a channel in order to pass it as a parameter to another routine, for example:

```
PROTOCOL PROT32 IS INT32 :
PROC p (CHAN OF INT32 X)
  X ! 99(INT32)
:
PROC q1 (CHAN OF PROT32 y)
  SEQ
    p (y)     -- this is illegal
    CHAN OF INT32 z RETYPES y :
    p(z)      -- this is legal
:
```

The facilities for **RETYPE**ing channels should only be used by programmers who understand the implementation of transputer channels, and the implications of attempting to circumvent occam's checking of channel usage. These facilities may be useful for those programmers who are using occam at a very low level, for example, writing loaders and other operating system type functions.

The current implementation of channels allows flexible use of channel arrays, which are implemented as an array of pointers to channel words. This means, for example, that it is possible to create an array of channels which map onto the hard links in a different order than 0 to 3, by using channel array constructors. For example:

```
CHAN OF ANY out.link0, out.link1, out.link2,
out.link3 :
PLACE out.link0 AT link0.out :
PLACE out.link1 AT link1.out :
PLACE out.link2 AT link2.out :
PLACE out.link3 AT link3.out :
[4]CHAN OF ANY outlink IS [out.link3, out.link1,
out.link2, out.link0] :
```

A particular effect of this implementation is that it may be useful to retype channels and arrays of channels into integers, in order to give the programmer access to these pointers. A programmer may set up an array of integers whose values are the addresses of channel words, and then use these as addresses of channels, like so:

```
[n]INT x:
SEQ
  ... initialize elements of array x, then:

  [n]CHAN OF protocol c RETYPES x:
  SEQ
    ... then communicate on c[i]
```

This will use the contents of `x[i]` as the address of the channel word. **Note:** channels set up in this way are not initialized automatically; you should initialize the contents of the channel word to `MOSTNEG INT` yourself, unless the channel word is mapped to a hard link.

Similarly channels may be retyped into pointers:

```
[n]CHAN OF protocol c :
SEQ
  VAL [n]INT x RETYPES c:
  SEQ i = 0 FOR n
    SEQ
      so.write.string (fs, ts, "The address of the
                              channel word of c[")
      so.write.int (fs, ts, i, 0)
      so.write.string (fs, ts, "] is : ")
      so.write.hex.int (fs, ts, x[i], 8)
      so.write.nl (fs, ts)
```

Note: retyping channels to pointers must be a `VAL RETYPE`. You may not modify the values of the pointers.

Single channels may be `RETYPED` to and from `INTs`.

Channel retyping should not be used to create arrays of existing channels. (See also section A.2.2 of the occam 2 *Toolset Language and Libraries Reference Manual*). Channel array constructors may be used for this:

```
PROC fancy.mux ([2]CHAN OF INT in, CHAN OF INT
                spare, out)
  [3]CHAN OF INT c IS [in[0], in[1], spare] :
  WHILE TRUE
    ALT i = 0 FOR 3
      INT data :
      c[i] ? data
        out ! data
:
```

13.3 Code insertion

This section describes the facilities provided by the occam 2 compiler code insertion mechanism using `ASM`.

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 optimized 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.

13.3.1 Using the code insertion mechanism

Code insertions may be introduced by the `ASM` construct. This section describes the use of the `ASM` construct. Details of the syntax are given in Appendix D.

The context of the `ASM` construct is determined, as with all occam constructs, by the text indentation. The transputer instructions which follow the `ASM` must be indented and there can only be one instruction per line. Lines may be terminated by a comment, which is introduced by a double dash ('--') as in occam. The transputer instructions are upper case versions of the standard mnemonics listed in *The transputer instruction set: a compiler writer's guide*.

Compiler options determine which instructions may be used within sections of code insertions, in the unit being compiled. The default is to disallow all code inserts. If the '`G`' option is used, then the instructions allowed are a restricted set of instructions which are sufficient for time-critical sections of sequential code. If the '`W`' option is used, then all transputer instructions are allowed. Since the inclusion of some instructions may have an unexpected effect on the occam program (for example, instructions which move the workspace pointer), instructions outside of the restricted set must be used with great care. Transputer instructions in the restricted set are listed in section C.8.

`ASM` statements can contain any number of primary or secondary transputer operations, transputer pseudo-operations, or labels.

In the transputer instruction set primary operations are *direct* instructions, *prefixing* instructions, or the special indirect instruction *opr*. Primary operations are always followed by an operand which can be any constant or constant expression. If additional *pfix* or *nfix* instructions are required to encode large values the `ASM` assembler automatically generates the required bytes. Secondary operations are any transputer *operation*, that is, any instruction selected using the *opr* instruction.

Pseudo-operations are more complex operations built up from sequences of instructions. Like macros, they expand into one or more transputer instructions, depending on their context and parameters.

For example, to perform a 1's complement addition we can write the following occam:

```
INT carry, temp:
SEQ
  carry, temp := LONGSUM (a, b, 0)
  c := carry PLUS temp
```

However, if this occurs in a time-critical section of the program we might replace it with:

```
ASM
  LDABC a, b, 0
  LSUM
  SUM
  ST c
```

which would avoid the storing and reloading of `carry` and `temp`. (**Note:** such examples are specific to the current compiler implementation; future releases are likely to behave differently).

Values in the range `MOSTNEG INT` to `MOSTPOS INT` may be used as operands to all of the direct functions without explicit use of prefix and negative prefix instructions. Access to non-local occam symbols is provided without explicit indirection, if you use the pseudo-instructions '`LD`', '`LDAB`' etc.

A more complex example, which sets an error if a value read from a channel is not in a particular range, takes advantage of both these facilities:

```
INT   a :
...  other code
PROC get.and.check.index (CHAN OF INT c)
  SEQ
    c ? a
    ASM
      LDAB 512, a     -- push value of free
                      -- variable onto stack
                      -- followed by 512
      CCNT1           -- if NOT (0 < a <= 512)
                      -- then set error
:
```

If there is a requirement for the code insertion to use some work space, then the work space may be declared before the `ASM` construct, in which case, the work space locations are accessed like any other occam symbol.

```
INT   a :
SEQ
  INT   b, c :
  ASM
    LD    a      -- push value in a onto stack
    ST    b      -- pop value from stack into b
    ...   more code
```

13.3.2 Special names

The following special names are available as constants inside **ASM** expressions.

`.WSSIZE`	Evaluates to the size of the current procedure's workspace. This will be the workspace offset of the return address, except within a replicated **PAR**, where it will be the size of that replication's workspace requirement.
`.VSPTR`	Evaluates to the workspace offset of the vector space pointer. When it is used inside a replicated **PAR**, it points to the vector space pointer for that branch only. A compile time error is generated if there is no vector space pointer because no vectors have been created.
`.STATIC`	Evaluates to the workspace offset of the static link. When it is used inside a replicated **PAR** it points to the static link for that branch only. A compile time error is generated if there is no static link.

For example, to determine the return address of a procedure, the following could be used: `LDL .WSSIZE`.

It is not checked that these names are used sensibly, for example, `J .WSSIZE` is legal even though it has no useful effect.

13.3.3 Labels and jumps

Labels may be defined inside an **ASM** construct. Labels are in scope for the entire procedure or function; thus both forward and backward references are permitted. It is illegal to declare two labels with the same name in the same routine.

To insert a label into the sequence of instructions, put the name of the label, preceded by a colon, on a line of its own. When the label is used in an instruction, the name is again preceded by a colon. For example:

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

Currently labels are declared in a different namespace from ordinary identifiers; thus it is possible to have both a label **x** and a variable **x** in scope at the same time;

the label is recognized in context (following a ':'). This will not necessarily be true in all implementations.

Branches may only be made to a label defined within the same procedure or function. It is permitted to branch to a PROC or FUNCTION which is in scope; it is up to the assembly programmer to load the parameters for the call correctly.

13.3.4 Workspace zero

Some transputer instructions make use of data at the current workspace pointer, known as '*workspace zero*'. These instructions are OUTBYTE, OUTWORD POSTNORMSN and the ALT disabling instructions.

If these instructions are used inside ASM, it is the programmer's responsibility to reserve this location by means of the allocation:

```
PLACE name AT WORKSPACE n :
```

See section 13.1.2.

13.3.5 Below workspace slots

Some instructions require various words at small negative offsets of workspace to be reserved. The compiler automatically reserves these when it sees the following instructions inside an ASM statement.

Instructions	Negative offsets
IN OUT OUTBYTE OUTWORD	3
ALT ALTWT ENBC ENBS	3
DISC DISS	3
VIN VOUT LDCNT	3
ENBG DISG GRANT	3
TIN TALT TALTWT ENBT DIST	5

13.3.6 Channels

Channels may be accessed in ASM; they are considered to be a pointer to a channel word. Thus 'loading' a channel will load a pointer to the channel word, and loading the 'address' of a channel will load a pointer to a pointer to the channel word.

13.3.7 Programming notes

1 Floating-point (fp) registers cannot be loaded directly; they must be loaded or stored by first loading a pointer to the register into an integer register and then using the appropriate floating-point instruction.

2 The operands to the load pseudo-ops must be small enough to fit in a register and the operands to the store pseudo-ops must be word-sized modifiable *elements*.

3 Code insertion using the `GUY` construct is obsolescent.

13.4 Dynamic code loading

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

These procedures are provided automatically by the compiler and are *not* referenced by a `#USE` directive. The procedures allow you to write an occam program that reads in a compiled occam procedure, and then calls it. The called procedure may be compiled and linked separately from the calling program and read in from a file. It is possible to pass parameters to the procedure, which must have at least 3 formal parameters.

Dynamically loadable code files can be created using the `icollect` '`K`' option. By default they are given the `.rsc` extension.

(Note that if you wish to dynamically load occam `FUNCTION`s, it is recommended that you call the `FUNCTION` indirectly from an occam `PROC`, and use non-`VAL` parameters to return the results to the calling environment).

The procedures for setting up parameters before the call and for making the call are outlined in the table below, and described in the following sections, with examples. Further information and examples of this technique can be found in section 5.3.5 of *The Transputer Applications Notebook – Systems and Performance.*

Procedure	**Parameter Specifiers**

`KERNEL.RUN`	`VAL []BYTE code,`
	`VAL INT entry.offset,`
	`[]INT workspace,`
	`VAL INT no.of.parameters`

`LOAD.INPUT.CHANNEL`	`INT here, CHAN OF ANY in`
`LOAD.INPUT.CHANNEL.VECTOR`	`INT here, []CHAN OF ANY in`
`LOAD.OUTPUT.CHANNEL`	`INT here, CHAN OF ANY out`
`LOAD.OUTPUT.CHANNEL.VECTOR`	`INT here, []CHAN OF ANY out`
`LOAD.BYTE.VECTOR`	`INT here, VAL []BYTE bytes`

The collector tool `icollect` can produce code in a format suitable for dynamic loading. The tool is described in Chapter 3 of the *occam 2 Toolset Reference Manual.*

13.4.1 Calling code

The occam 2 compiler recognizes calls of a procedure `KERNEL.RUN` with the following parameters:

```
PROC KERNEL.RUN (VAL []BYTE code,
                 VAL INT entry.offset,
                 []INT workspace,
                 VAL INT no.of.parameters)
```

The effect of this procedure is to call the procedure loaded in the `code` buffer, starting execution at the location `code[entry.offset]`.

The `code` to be called must begin at a word-aligned address. To ensure proper alignment either start the array at zero or realign the code on a word boundary before passing it into the procedure.

The `workspace` buffer is used to hold the local data of the called procedure. For details of the contents of the `workspace` buffer see Figure 13.1. The required size of this buffer and the code buffer must be derived from information in the code file.

The parameters passed to the called procedure should be placed at the top of the `workspace` buffer by the calling procedure before the call of `KERNEL.RUN`. The call to `KERNEL.RUN` returns when the called procedure terminates. If the called procedure requires a separate vector space, then another buffer of the required size must be declared, and its address placed as the last parameter at the top of `workspace`. As calls of `KERNEL.RUN` are handled specially by the compiler it is necessary for `no.of.parameters` to be a constant known at compile time and to have a value ≥ 3.

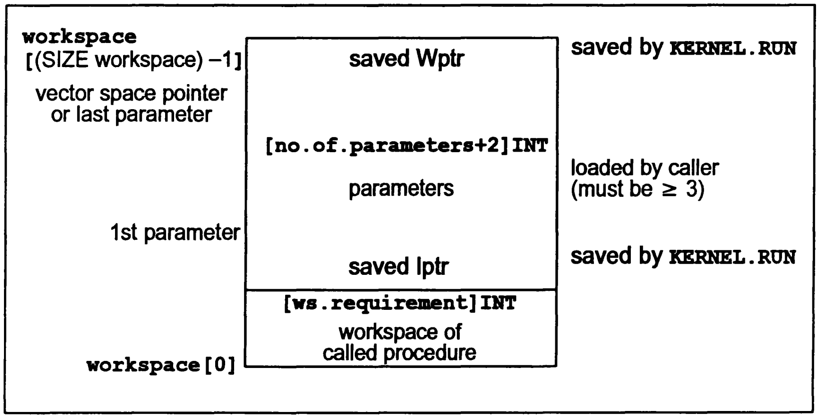


Figure 13.1 Workspace buffer

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

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

where `ws.requirement` is the size of workspace required, determined when the called procedure was compiled and stored in the code file, and `no.of.parameters` includes the vector space pointer if it is required. The parameters must be loaded before the call of `KERNEL.RUN`. The parameter corresponding to the first formal parameter of the procedure should be in the word adjacent to the saved `Iptr` word, and the vector space pointer or the last parameter should be adjacent to the top of workspace where the `Wptr` word will be saved.

13.4.2 Loading parameters

There are a number of library procedures to set up parameters before the call. These are:

```
LOAD.INPUT.CHANNEL (INT here, CHAN OF ANY in)
```

The variable `here` is assigned the address of the input channel `in`.

```
LOAD.INPUT.CHANNEL.VECTOR (INT here,
                           []CHAN OF ANY in)
```

The variable `here` is assigned the address of the base element of the channel array `in` (i.e. the base of the array of pointers).

```
LOAD.OUTPUT.CHANNEL (INT here, CHAN OF ANY out)
```

The variable `here` is assigned the address of the output channel `out`.

```
LOAD.OUTPUT.CHANNEL.VECTOR (INT here,
                            []CHAN OF ANY out)
```

The variable `here` is assigned the address of the base element of the channel array `out` (i.e. the base of the array of pointers).

```
LOAD.BYTE.VECTOR (INT here, VAL []BYTE bytes)
```

The variable `here` is assigned the address of the byte array `bytes`.

Note that when passing vector parameters, if the formal parameter of the `PROC` called is *unsized* then the vector address must be followed by the number of elements in the vector, for example:

```
LOAD.BYTE.VECTOR(param[0], buffer)
param[1] := SIZE buffer
```

Thus an unsized vector parameter requires 2 parameter slots. The size must be in the units of the array (not in bytes, unless it is a byte vector, as above). For multidimensional 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.

13.4.3 Examples

This section gives two examples of dynamic loading. The first is a simple example showing how parameterless code can be input on a channel and loaded. The second is a more complex example showing how to set up and pass parameters into a dynamically loaded program. Sources can be found in the `examples/manuals/dynamic` directory.

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 work-space 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 SP fs, ts, []INT buffer,
              VAL BOOL debugging, INT result)
```

The two channel parameters **fs** and **ts** handle output from and input to the file server; the **INT** vector acts as a buffer. The two channels and the buffer are the same parameters as are provided by the bootstrap code added by the collector tool (see Chapter 3 in the *occam 2 Toolset Reference Manual*), 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.

```
#INCLUDE "hostio.inc"
PROC call.program (CHAN OF SP fs, ts, []INT free.memory)

  -- Variables for holding code and entry and workspace
  -- data read from file
  [256]BYTE code:
  INT code.length, entry.offset, work.space.size:
  INT vector.space.size:
  INT result:              -- Variable used by process
  VAL debugging IS TRUE:   -- Value param for process
  VAL no.params IS 7:      -- No. of parameter slots
  -- Need 1 slot per parameter + 1 for the size of the
  -- array parameter + 1 for the vector space pointer

  SEQ
    -- Read in code and data about code

    -- Slice up memory vector for use by process
    -- Reserve work space requirement for process
    []INT ws IS [free.memory FROM 0 FOR
            work.space.size + (no.params + 2)]:
    -- Reserve vector space requirement for process
    []INT vs IS [free.memory FROM SIZE ws FOR
                 vector.space.size]:
    -- Reserve remainder of memory for use
    -- as process parameter buffer
    []INT buffer IS
      [free.memory FROM (SIZE ws) + (SIZE vs) FOR
       (SIZE free.memory) - ((SIZE ws) + (SIZE vs))]:
    SEQ
      -- Reserve slot in ws for parameters
      []INT parameter IS
        [ws FROM work.space.size + 1 FOR no.params]:
      SEQ
        LOAD.INPUT.CHANNEL (parameter[0], fs)
        LOAD.OUTPUT.CHANNEL(parameter[1], ts)
        -- Retype buffer to take its address
        []BYTE b.buffer RETYPES buffer:
        LOAD.BYTE.VECTOR(parameter[2], b.buffer)
        parameter[3] := SIZE buffer
```

```
          -- Store VAL BOOL parameter
          parameter[4] := INT debugging
          -- Store address of INT parameter
          []BYTE b.result RETYPES result:
          LOAD.BYTE.VECTOR(parameter[5], b.result)
          -- Store pointer to vector space
          []BYTE b.vs RETYPES vs:
          LOAD.BYTE.VECTOR(parameter[6], b.vs)
        -- Run the process
        KERNEL.RUN([code FROM 0 FOR code.length],
                   entry.offset, ws, no.params)
  :
```

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 13.2.

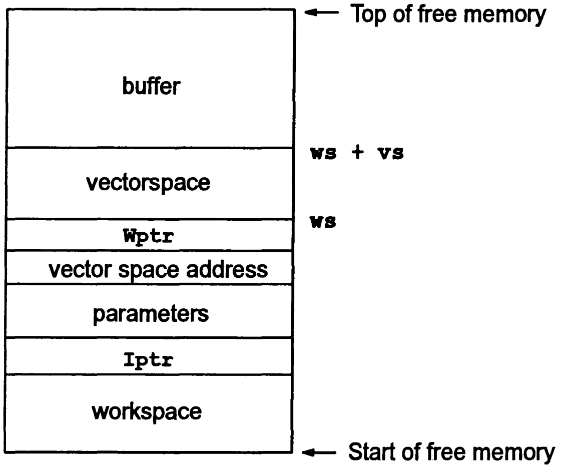


Figure 13.2 Partitioning of free memory

13.5 Extraordinary use of links

13.5.1 Introduction

The transputer link architecture provides ease of use and compatibility across the range of transputer products. It provides synchronized 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 synchronized 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 1.10 in the *occam 2 Toolset Language and Libraries Reference Manual.*

13.5.2 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 analyze 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.

13.5.3 Programming concerns

The first concern of a designer is to understand how to recognize 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 reinitializing the link hardware; again there is a suitable library procedure to allow this to be performed.

13.5.4 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 recognized. Two procedures provide input and output where communication failure can be detected by a simple timeout, the other two procedures provide input and output where the failure of the communication is signalled to the procedure via a channel. The procedures have a boolean variable as a parameter which is set `TRUE` if the procedure terminated as a result of communication failure being detected, and is set `FALSE` otherwise. If the procedure does terminate as a result of communication failure then the link channel can be reset.

All four library procedures take as parameters a link channel `c` (on which the communication is to take place), a byte vector `mess` (which is the object of the communication) and the boolean variable `aborted`. The choice of a byte vector as the parameter to these procedures allows an object of any type to be passed along the channel provided it is retyped first. Channel retyping (see section 13.2) may be used to pass channels of any protocol to these procedures.

The two procedures for communication where failure is detected by a timeout take a timer parameter `TIME`, and an absolute time `t`. The procedures treat the communication as having failed when the time as measured by the timer `TIME` is `AFTER` the specified time `t`. The names and the parameters of the procedures are as follows:

```
InputOrFail.t(CHAN OF ANY c, []BYTE mess,
              TIMER TIME,
              VAL INT t, BOOL aborted)

OutputOrFail.t(CHAN OF ANY c, VAL []BYTE mess,
               TIMER TIME,
               VAL INT t, BOOL aborted)
```

The other two procedures provide communication where failure cannot be detected by a simple timeout. In this case failure must be signalled to the inputting or outputting procedure via a message on the channel `kill`. The message is of type `INT`. The names and parameters to the procedures are as follows:

```
InputOrFail.c(CHAN OF ANY c, []BYTE mess,
              CHAN OF INT kill, BOOL aborted)

OutputOrFail.c(CHAN OF ANY c, VAL []BYTE mess,
               CHAN OF INT kill, BOOL aborted)
```

13.5.5 Recovery from failure

To reuse a link after a communication failure has occurred it is necessary to reinitialize the link hardware. This involves reinitializing both ends of both channels implemented by the link. Furthermore, the reinitialization 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 reinitialize 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 behavior is undefined.

13.5.6 Example: a development system

For our example consider the development system illustrated in Figure 13.3.

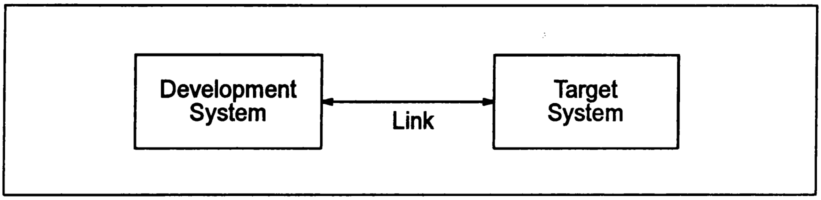


Figure 13.3 Development system

The first step in the solution is to recognize that the development system knows when a failure might occur, and hence knows when it might be necessary to abort a communication.

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 reinitialize 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 reinitialized.

```
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 `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.

13.6 Scheduling

Processes in occam may have one of two priorities, high or low. A high priority process will be executed in preference to a low priority process if both are active, so that a low priority process will be interrupted. The `PRI PAR` construct is used to assign priority to processes.

Scheduling in occam is achieved using the transputer's scheduler which maintains a list of processes. The following predefined procedure may be used to affect scheduling:

- `RESCHEDULE()` – inserts instructions into the program to cause the current process to be moved to the end of the current priority scheduling queue, even if the current process is a 'high priority' process.

This procedure is recognized automatically by the compiler and does not need to be referenced by the `#USE` directive.

13.7 Setting the error flag

The transputer error flag can be explicitly set from software using the following predefined procedure:

- `CAUSEERROR()` – inserts a *seterr* instruction into the program. If the program is in STOP or UNIVERSAL mode it inserts a *stopp* instruction as well.

This procedure is recognized automatically by the compiler and does not need to be referenced by the `#USE` directive.

`CAUSEERROR` sets the transputer error flag no matter what the error mode of the compilation. This is distinct from the occam primitive process `STOP`, which only sets the flag if the compilation is in HALT mode.

Appendices

A Configuration language definition

This appendix defines the syntax of the occam configuration language. This should be considered as extending the syntax of occam.

A.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly, the form is as follows:

- Each phrase definition consists of an equality expression built up using an equals sign to separate the two sides.
- Terminal strings of the language — those not built up by rules of the language — are printed in teletype font e.g. `NODE`.
- Alternatives are separated by vertical bars ('|').
- Optional sequences are enclosed in italic square brackets ('*[*' and '*]*').
- Items which may be repeated zero or more times appear in braces ('{' and '}').
- $\{_0, x\}$ represents a list of zero or more items of type 'x' separated by commas.
- $\{_1, x\}$ represents a list of one or more items of type 'x' separated by commas.

A.2 Introduction

A configuration program file contains a sequence of specifications. These specifications should include one hardware description (containing at least one node declaration), and one software description. Optionally there may be edge declarations and arc declarations. An optional mapping may appear either before of after the software configuration, but after the declaration of any nodes, edges or arcs which it references. Normal occam scope rules apply.

The `#INCLUDE` mechanism may be used to incorporate hardware descriptions, software descriptions, or any other source text from other files.

The `#USE` statement may be used to reference pre-compiled code, either at the outer level, or within the software description.

A.3 New types and specifications

This section defines the new occam types introduced by the configuration language.

The syntax adds the new primitive types **NODE**, **EDGE** and **ARC**, and structures **CONFIG**, **NETWORK** and **MAPPING** to the occam language.

NODE declarations introduce processors (*nodes* of a graph). These processors are *physical* if their type and memory size attributes are defined as part of the hardware description, and *logical* otherwise.

EDGE declarations introduce external connections of the hardware description.

ARC declarations introduce named connections (*arcs* of a graph). Each arc connects two edges, which may be attributes of nodes, or declared edges. Connections need only be named if it is required to force a particular mapping of channels, or if names are required to aid debugging.

NETWORK declarations introduce the hardware description.

CONFIG declarations introduce the software description.

MAPPING declarations introduce the mapping description.

A.3.1 Syntax of configuration description

```
configuration      =    hardware.description
                        software.description
                        [ mapping ]
                   |    specification
                        configuration

specification           VAL

primitive type     =    NODE
                   =    EDGE
                   =    ARC
```

A.4 Hardware description

The **NETWORK** keyword introduces a hardware description, an optionally named structure which describes the types, connectivity and attributes of previously declared processor nodes. Connections are defined in **CONNECT** statements. Attributes are given values in **SET** statements. The attributes of a processor node include an array of edges which are its links, a string which defines its processor type, and an integer which is the memory size in bytes.

Connections and attribute settings may be combined in any order using the **DO** constructor, including replication and conditionals. For each node which has a type defined to be a processor the attributes with predefined names `type` and `memsize` must be set once only. The connections connect declared edges and edges of nodes, which have the predefined attribute name `link`. The boolean attribute `root` may be set to `TRUE` for only one node in a network without a connection to the predefined edge `HOST`. The attribute `romsize` defines the size in bytes of read only memory on a node. Attributes are referenced by subscripting node names with attribute names in brackets.

A.4.1 Processor attributes

This section describes processor attributes defined in the occam configuration language.

The following processor attributes can be defined in the `NETWORK` description:

- `link` – used by processor and network nodes to define interconnection. Only defined if the `type` attribute has already been defined.
- `memsize` – used by processor nodes to define memory size.
- `root` – defines the root processor if there is no host connection.
- `romsize` – specifies the size of ROM attached to the processor, expressed as an integer.
- `type` – used by processor nodes to define processor type.

The following processor attributes can be defined in the `MAPPING` description:

- `linkquota` – suggests the maximum number of links on the associated processor that should be used by the virtual channel routing system.
- `location.code`, `location.ws`, `location.vs`, – used by process nodes to specify the absolute locations of their code, workspace, and vectorspace segments.
- `order.code`, `order.ws`, `order.vs` – used to specify the ordering of code and data segments.
- `nodebug` – disables debugging by the *Inquest Toolset*. Takes the value `TRUE` or `FALSE`; the default is `FALSE`.
- `noprofile` – disables debugging by the *Inquest Toolset*. Takes the value `TRUE` or `FALSE`; the default is `FALSE`.
- `reserved` – used by processor nodes to reserve memory.
- `routecost` – defines within the range 1 to 1000000 the associated cost of through-routing data through this processor for other processor's virtual channel traffic.

- `tolerance` – controls with any value between 0 and 1000000 how much the particular processor concerned can be used for the provision of load-sharing through-routing paths for other processors.

A.4.2 Syntax definition

```
hardware.description      =   NETWORK [ networkname ]
                                 network.item
                                 :
                          |   specification
                              hardware.description

specification             =   node.declaration
                          |   edge.declaration
                          |   arc.declaration
                          |   VAL

node.declaration          =   {0 [ expression ] } NODE nodename :
edge.declaration          =   {0 [ expression ] } EDGE edgename :
arc .declaration          =   {0 [ expression ] } ARC arcname :

network.item              =   DO
                          |   DO replicator
                                 network.item
                          |   connection.item
                          |   attribute.setting
                          |   conditional.network.item
                          |   SKIP
                          |   STOP
                          |   abbreviation
                              network.item

connection.item           =   CONNECT edge TO edge [ WITH arcname ]

edge                      =   edgename
                          |   device "[link]" "["expression"]"

attribute.setting         =   SET device { attribute.assignment }
device                    =   nodename {0 [ subscript ] }
attribute.assignment      =   {1 , attribute } := {1 , attribute.value }
attribute                 =   attribute.name {0 , [ subscript ] }
attribute.value           =   expression

conditional.network.item  =   IF
                                 {network choice}

network choice            =   guarded.network.choice
                          |   conditional.network.item

guarded .network.choice   =   boolean
                              network.item
```

A.5 Software description

A **CONFIG** declaration introduces the software description as an occam process. Additional specifications and processes are added to occam: the processor name in a **PROCESSOR** statement may be a physical processor name or the name of a logical processor which is mapped onto a physical processor. A channel allocation may allocate up to two channels onto a named arc of the network.

A.5.1 Syntax definition

```
software.description       =    CONFIG [config.name]
                                  placedpar
                                :
                           |    specification
                                software.description

placedpar                  =    [ PLACED ] PAR
                                  { placedpar }
                           |    [ PLACED ] PAR replicator
                                  placedpar
                           |    PROCESSOR processor.name
                                  process.name
                           |    specification
                           |    placedpar
                           |    conditional.network.item
                           |    SKIP
                           |    STOP

conditional.network.item   =    IF
                                  {network choice}

network choice             =    guarded.network.choice
                           |    conditional.network.item

guarded .network.choice    =    boolean
                                network.item
```

A.6 Mapping structure

The keyword **MAPPING** introduces an (optionally named) mapping structure which may be either before or after the software description. **DO** and **IF** constructs may be used as in occam. **SKIP** and **STOP** are also allowed.

Mappings are introduced by the **MAP** keyword. A mapping may be used to associate logical processors with physical processors (code mapping), and channels with arcs (channel mappings).

In code mappings a logical processor may appear on the left hand side of only one mapping item whereas a physical processor may appear on the right hand side of more than one mapping item. A code mapping may include a priority clause, introduced by `PRI`, which will determine the priority at which the process will run. `SET` may be used to define processor attributes (see section A.4.1).

In channel mappings the arc must connect the nodes onto which the processes using the channels are mapped. The effect of channel mappings is identical to the corresponding channel allocations which may appear in the software description. Mappings involving single channels may use the `PLACE` statement.

Channel mappings are optional except in the case where one end of the arc is an external edge. (The configurer will normally choose a mapping from its knowledge of the connectivity of the hardware and the implied connectivity derived from the use of channels as in the software description.)

A.6.1 Syntax definition

```
mapping              =   MAPPING [ mapping.name ]
                            mapping.item
                          :
                     |   specification
                         mapping

mapping.item         =   code.mapping
                     |   channel.mapping
                     |   channel.allocation
                     |   DO
                            mapping.item
                     |   DO replicator
                            mapping.item
                     |   conditional.map.item
                     |   SKIP
                     |   STOP
                     |   abbreviation
                         mapping.item
                     |   attribute.setting


code.mapping         =   MAP processor.list ONTO node [priority.clause]
priority.clause      =   PRI expression
processor.list       =   {1 , processid }
processid            =   processor.name {0 [ subscript ] }
processor.name       =   node.name
node                 =   node.name {0 [ subscript ] }


channel.mapping      =   MAP channel.list ONTO arc
channel.list         =   {1 , channelid }
```

channelid	=	*channel.name* {$_0$ [*subscript*] }
arc	=	*arc.name* {$_0$ [*subscript*] }
channel.allocation	=	**PLACE** { *channel.list* } **ON** *arc* :
attribute.setting	=	**SET** *device* { *attribute.assignment* }
device	=	*nodename* {$_0$ [*subscript*] }
attribute.assignment	=	{$_1$, *attribute* } := {$_1$, *attribute.value* }
attribute	=	*attribute.name* {$_0$, [*subscript*] }
attribute.value	=	*expression*
conditional.map.item	=	**IF**
		{*mapping.choice*}
mapping.choice	=	*guarded.mapping.choice*
	|	*conditional.map.item*
guarded.mapping.choice	=	*boolean*
		map.item

A.7 Constraints

The following constraints apply to all configurations:

- All physical processors whose `type`s are set must be connected to each other.
- Any physical processor whose `type` is set must have its `memsize` set.
- Logical processors may only be mapped onto physical processors whose `type` has been set.
- Channels connecting processors of different word size must not use protocols based on the type `INT`.
- A priority expression must evaluate to 0 (high) or 1 (low).
- `INT` array constructors such as `[1,2,3]` are not accepted for 16-bit processors. They should be converted into `INT16` arrays.
- It is not permitted to `RETYPE INT` constants into other types for 16-bit processors. `INT16` constants should be used instead.
- `INT` expressions are treated as `INT32`s. Thus `MOSPOS INT` evaluates to the same value as `MOSTPOS INT32`. Where this is a problem, i.e. it causes a 16-bit integer overflow, the configurer will generate an error.

B Equivalent data types

This appendix lists equivalent data types to use when passing parameters to external routines and receiving function return values. The information is presented with both occam and C as the calling language.

B.1 occam as the calling language

B.1.1 Parameter passing

The following table lists the equivalent data types to use when passing parameters from occam to C. The first column gives the C formal parameter, the second and third columns give the occam actual parameter type to pass. Where there is no true equivalent the action to take is given.

C formal parameter	occam actual parameter	
	(32 bit)	(16 bit)

`char`
`unsigned char`
	`VAL BYTE`	`VAL BYTE`

`signed char`	No direct equivalent†	No direct equivalent†

`short`
`signed short`
	No direct equivalent†
	(see Note 1)
		`VAL INT`
		`VAL INT16`

`unsigned short`	No direct equivalent†	No direct equivalent†

`int`
`signed int`
`enum`
	`VAL INT`
	`VAL INT32`
		`VAL INT`
		`VAL INT16`

`unsigned int`	No direct equivalent†	No direct equivalent†

`long`
`signed long`
	`VAL INT`
	`VAL INT32`
		No direct equivalent†

`unsigned long`	No direct equivalent†	No direct equivalent†
`float`	`VAL REAL32`	No direct equivalent†
`double`	No direct equivalent†	No direct equivalent†
`struct` `union`	No direct equivalent†	No direct equivalent†

`char *`
`unsigned char *`
	`BYTE`	`BYTE`

`signed char *`	No direct equivalent†	No direct equivalent†

`short *`
`signed short *`
	`INT16`	`INT16`
		`INT`

`unsigned short *`	No direct equivalent†	No direct equivalent†

`int *`
`signed int *`
`enum *`
	`INT`
	`INT32`
		`INT`
		`INT16`

`unsigned int *`	No direct equivalent†	No direct equivalent†

`long *`
`signed long *`
	`INT`
	`INT32`
		`INT32`

`unsigned long *`	No direct equivalent†	No direct equivalent†
`float *`	`REAL32`	`REAL32`
`double *`	`REAL64`	`REAL64`

`struct *`
`union *`
	No direct equivalent†	No direct equivalent†

`channel *`	`CHAN`	`CHAN`
array	See section 11.1.4.	See section 11.1.4.
†There is no direct type equivalent in occam. Either recode the C program or pass the parameter in another form. **Note 1**: A C `short` on a 32 bit processor is stored in 32 bits with the upper 16 bits zeroed. In occam an `INT16` on a 32 bit processor is also stored as a 32 bit value, however, in this case the upper 16 bits are ignored and not zeroed. Hence C `short` and occam `INT16` are not directly equivalent.		

B.1.2 Return values

The following table gives the occam data type to use when receiving return values from C functions. Equivalents are given separately for 32 bit and 16 bit transputers.

C function type	occam function type	
	(32 bit)	(16 bit)

`char`
`unsigned char`
	`BYTE`	`BYTE`

`signed char`	No direct equivalent†	No direct equivalent†

`short`
`signed short`
	`INT16`	`INT`
		`INT16`

`unsigned short`	No direct equivalent†	No direct equivalent†

`int`
`signed int`
`enum`
	`INT`
	`INT32`
		`INT`
		`INT16`

`unsigned int`	No direct equivalent†	No direct equivalent†

`long`
`signed long`
	`INT`
	`INT32`
		`INT32`

`unsigned long`	No direct equivalent†	No direct equivalent†
`float`	**REAL32**	**REAL32**
`double`	**REAL64**	**REAL64**
`struct` `union`	No direct equivalent†	No direct equivalent†
Any pointer type	No direct equivalent†	No direct equivalent†
†There is no direct type equivalent in occam. Either recode the C program or pass the parameter in another form.		

B.1.3 Example of passing parameters from occam to C

The following examples show two C functions with a variety of formal parameters along with the occam code which can call them. The code for 32 bit and 16 bit transputers is given separately.

C functions to be called on a 32-bit transputer:

```
int cfuncl(int parml);

#pragma IMS_nolink(cfuncl)      /* remove the gsb hidden
                                   parameter */

void cprocl(char c, int i,
            long l, float f,
            char *cp, short *sp,
            int *ip, long *lp,
            float *fp, double *dp,
            int array1[8],
            int array2[], const int array2len);

#pragma IMS_nolink(cprocl)      /* remove the gsb hidden
                                   parameter */

int cfuncl(int parml)
{
  return parml * 10;
}

void cprocl(char c, int i,
            long l, float f,
            char *cp, short *sp,
            int *ip, long *lp,
            float *fp, double *dp,
            int array1[8],
            int array2[], const int array2len)
{
  int j;

  *cp = c;
  *sp = (short)c;
  *ip = i;
  *lp = l;
```

```
    *fp = f;
    *dp = (double)i;
    for (j = 0; j < 8; j++)
      array1[j] = 42;
    for (j = 0; j < array2len; j++)
      array2[j] = array2len;
  }
```

occam code to call the above C functions on a 32 bit transputer:

```
  #PRAGMA EXTERNAL "INT FUNCTION cfunc1(VAL INT parm1) = 100"

  #PRAGMA EXTERNAL "PROC cproc1(VAL BYTE c, VAL INT i, *
                            * VAL INT32 l, VAL REAL32 f, *
                            * BYTE cp, INT16 sp, *
                            * INT ip, INT32 lp, *
                            * REAL32 fp, REAL64 dp, *
                            * [8]INT array1, []INT array2) = 100"

    BYTE c, cp:
    INT i, ip, result:
    INT16 sp:
    INT32 l, lp:
    REAL32 f, fp:
    REAL64 dp:
    [8]INT array1:
    [5]INT array2:
    SEQ
      result := cfunc1(i)
      cproc1(c, i, l, f, cp, sp, ip, lp, fp, dp, array1, array2)
```

C functions to be called on a 16 bit transputer:

```
  int cfunc1(int parm1);

  #pragma IMS_nolink(cfunc1)     /* remove the gsb hidden
                                    parameter */

  void cproc1(char c, int i,
              short s, char *cp,
              short *sp, int *ip,
              long *lp, float *fp,
              double *dp, int array1[8],
              int array2[], const int array2len);

  #pragma IMS_nolink(cproc1)     /* remove the gsb hidden
                                    parameter */

  int cfunc1(int parm1)
  {
    return parm1 * 10;
  }

  void cproc1(char c, int i,
              short s, char *cp,
              short *sp, int *ip,
              long *lp, float *fp,
               double *dp, int array1[8],
               int array2[], const int array2len)
{
  int j;

  *cp = c;
  *sp = s;
  *ip = i;
  *lp = (long)i;
  *fp = (float)i;
  *dp = (double)i;
  for (j = 0; j < 8; j++)
    array1[j] = 42;
  for (j = 0; j < array2len; j++)
    array2[j] = array2len;
}
```

occam code to call the above C functions on a 16 bit transputer:

```
#PRAGMA EXTERNAL "INT FUNCTION cfunc1(VAL INT parm1) = 100"

#PRAGMA EXTERNAL "PROC cproc1(VAL BYTE c, VAL INT i, *
                              * VAL INT16 s, BYTE cp, *
                              * INT16 sp, INT ip, *
                              * INT32 lp, REAL32 fp, *
                              * REAL64 dp, *
                              * [8]INT array1, []INT array2) = 100"

  BYTE c, cp:
  INT i, ip, result:
  INT16 s, sp:
  INT32 lp:
  REAL32 fp:
  REAL64 dp:
  [8]INT array1:
  [5]INT array2:
  SEQ
    result := cfunc1(i)
    cproc1(c, i, s, cp, sp, ip, lp, fp, dp, array1, array2)
```

B.2 C as the calling language

B.2.1 Parameter passing

The following table lists the equivalent data types to use when passing parameters from C to occam. The first column gives the occam formal parameter, the second and third columns give the C actual parameter type to pass. Where there is no true equivalent the action to take is given.

occam formal parameter	C actual parameter	
	(32 bit)	(16 bit)
VAL BOOL	int (value must be 0 or 1)	int (value must be 0 or 1)
VAL BYTE	char unsigned char	char unsigned char
VAL INT16	short int	short int int
VAL INT32	int long int	long int *
VAL INT64	No direct equivalent†	No direct equivalent†
VAL INT	int	int
VAL REAL32	float	float *
VAL REAL64	double *	double *
VAL *array*	See section 11.1.4.	See section 11.1.4.
BOOL	char * unsigned char * (value pointed to must be 0 or 1)	char * unsigned char* (value pointed to must be 0 or 1)
BYTE	char * unsigned char *	char * unsigned char *
INT16	short int *	short int * int *
INT32	int * long int *	long int *
INT64	No direct equivalent†	No direct equivalent†
INT	int *	int *
REAL32	float *	float *
REAL64	double *	double *
CHAN	Channel * (see Note 1)	Channel * (see Note 1)
PORT	No direct equivalent†	No direct equivalent†
TIMER	Pass nothing (see page 203).	Pass nothing (see page 203).
array	See section 11.1.4.	See section 11.1.4.

†There is no direct type equivalent in C. Either recode the occam program or pass the parameter in another form.

Note 1: Channel is an INMOS specific type declared in the C header file channel.h.

B.2.2 Return values

The following table outlines the conventions that must be followed when receiving occam function return values in C.

occam function type	C function type	
	(32 bit)	(16 bit)
`BOOL`	`int`	`int`

`BYTE`	`char`
	`unsigned char`
		`char`
		`unsigned char`

`INT16`	`short int`	`short int`
		`int`

`INT32`	`int`
	`long int`
		`long int`

`INT64`	No direct equivalent†	No direct equivalent†
`INT`	`int`	`int`
`REAL32`	`float`	`float`
`REAL64`	`double`	`double`
†There is no direct type equivalent in C. Either recode the occam program or pass the parameter in another form.		

B.2.3 Example of passing parameters

The following example shows an occam function and an occam procedure with a variety of formal parameters, along with the C code which can call them. The calling code for 32 bit and 16 bit transputers is given separately. The occam routines to be called are as follows:

```
INT32 FUNCTION ocfunc1(VAL INT32 parm1) IS parm1:

PROC ocproc1(VAL BYTE vb,
             VAL INT16 vi16,
             VAL INT32 vi32,
             VAL INT vi,
             VAL REAL32 vr32,
             VAL REAL64 vr64,
             VAL BOOL vbo,
             VAL []INT varr1,
             VAL [8]INT varr2,
             BYTE b,
             INT16 i16,
             INT32 i32,
             INT i,
             REAL32 r32,
             REAL64 r64,
             BOOL bo,
             []INT arr1,
             [8]INT arr2)
```

```
    SEQ
      b := vb
      i16 := vi16
      i32 := vi32
      i := vi
      r32 := vr32
      r64 := vr64
      bo := vbo
      arr1 := varr1
      arr2 := varr2
  :
```

The C code to call the above occam routines on a 16 bit transputer is as follows:

```
#define ARRAY_SIZE_1 4
#define ARRAY_SIZE_2 8

extern long int ocfunc1(long int parm1);

extern void ocproc1(char vb, short int vi16,
                    long int vi32, int vi,
                    float vr32, double *vr64,
                    int vbo,
                    int varr1[], const int varr1_size,
                    int varr2[ARRAY_SIZE_2],
                    char *b, short int *i16,
                    long int *i32, int *i,
                    float *r32, double *r64,
                    char *bo,
                    int arr1[], const int arr1_size,
                    int arr2[ARRAY_SIZE_2]);

#pragma IMS_nolink(ocfunc1)
#pragma IMS_nolink(ocproc1)

long int li, result;
char vb, b;
short int vi16, i16;
long int vi32, i32;
int vi, i;
float vr32, r32;
double vr64, r64;
int vbo;
char bo;
int varr1[ARRAY_SIZE_1], arr1[ARRAY_SIZE_1];
int varr2[ARRAY_SIZE_2], arr2[ARRAY_SIZE_2];

result = ocfunc1(li);

ocproc1(vb, vi16, vi32, vi, vr32, &vr64,
        vbo, varr1, ARRAY_SIZE_1, varr2,
        &b, &i16, &i32, &i, &r32, &r64,
        &bo, arr1, ARRAY_SIZE_1, arr2);
```

The C code to call the above occam routines on a 32 bit transputer is as follows:

```
#define ARRAY_SIZE_1 4
#define ARRAY_SIZE_2 8

extern long int ocfuncl(long int *parml);

extern void ocprocl(char vb, short int vil6,
                    long int *vi32, int vi,
                    float *vr32, double *vr64,
                    int vbo,
                    int varrl[], const int varrl_size,
                    int varr2[ARRAY_SIZE_2],
                    char *b, short int *il6,
                    long int *i32, int *i,
                    float *r32, double *r64,
                    char *bo,
                    int arrl[], const int arrl_size,
                    int arr2[ARRAY_SIZE_2]);

#pragma IMS_nolink(ocfuncl)
#pragma IMS_nolink(ocprocl)

long int li, result;
char vb, b;
short int vil6, il6;
long int vi32, i32;
int vi, i;
float vr32, r32;
double vr64, r64;
int vbo;
char bo;
int varrl[ARRAY_SIZE_1], arrl[ARRAY_SIZE_1];
int varr2[ARRAY_SIZE_2], arr2[ARRAY_SIZE_2];

res = ocfuncl(&li);

ocprocl(vb, vil6, &vi32, vi, &vr32, &vr64,
        vbo, varrl, ARRAY_SIZE_1, varr2,
        &b, &il6, &i32, &i, &r32, &r64,
        &bo, arrl, ARRAY_SIZE_1, arr2);
```

C Transputer instruction set

This appendix provides a reference for the transputer instruction set as supported by assembly code insertion. For detailed specifications of the instructions, refer to the *Transputer instruction set: a compiler writer's guide.*

Instructions listed in sections C.1 to C.7 can be used when 'full code insertion' is enabled by the 'W' command line option. Instructions listed in section C.8 can be used when 'sequential code insertion' is enabled by the 'G' command line option. **Note**: Only use those instructions which exist on the target processor may be used. For example, floating point instructions (see section C.5.1) may not be used on T4 series transputers.

C.1 Prefixing instructions

The transputer instruction set is built up from 16 *direct* instructions, each with a 4-bit argument field. The direct instructions include *prefix* instructions which augment the 4-bit field in a direct instruction which follows them by their own 4-bit argument field, effectively allowing the argument to be extended to 32 bits. Normally, the assembler will compute the prefix instructions required for operand values greater than 4 bits automatically.

pfix	prefix
nfix	negative prefix

C.2 Direct instructions

The direct instructions form the core of the transputer instruction set. Each direct instruction has a single operand, normally an integer constant, which will be encoded in the instruction itself and, if it is larger than will fit into the 4-bit argument field of the direct instruction, into a series of *pfix* and *nfix* instructions as well.

The transputer architecture is based around a three-register *evaluation stack* and a single base register **Wreg**. The load and store 'local' instructions access a word in memory at a displacement from **Wreg** given by the operand value used. The displacement is scaled by the word size. The load and store 'non-local' instructions use the top evaluation stack register (**Areg**) as the base instead of **Wreg**, allowing computed base addresses to be used.

The operand of the *j*, *cj* and *call* instructions is interpreted as a byte displacement from the instruction pointer (program counter) register **Iptr**. *ldpi* is similar but takes its operand from **Areg**.

adc	Add constant operand value to **Areg**.
ajw	Adjust workspace pointer **Wreg** by constant operand value (scaled by word length).
call	Call.
cj	Conditional jump i.e. 'jump if zero otherwise pop **Areg**'. As with *jump*, a label identifier may be used as argument to this instruction.
eqc	Test if **Areg** equals constant; (result 'true' or 'false', placed in **Areg**).
j	Jump: the argument may be an identifier indicating a label for the jump to go to; the assembler will compute the displacement required.
ldc	Load constant.
ldl	Load local word
ldlp	Load pointer to local word
ldnl	Load non-local word
ldnlp	Load pointer to non-local word
opr	'operate': the argument to this instruction is a code indicating a zero-operand *indirect* instruction to be executed. Most of the transputer instruction set is made up of these indirect instructions. Normally you would use the mnemonic for the specific indirect instruction which you require: the assembler will encode this as an *opr* instruction on your behalf. However, it is possible to use *opr* explicitly, for example to synthesize the instruction sequence for a new indirect instruction not supported by the T414 and T800 transputers.
stl	Store local word
stnl	Store non-local word

C.3 Operations

The instructions in this section are all *indirect* instructions built out of the *opr* instruction. None of these instructions take an argument; instead, they work solely with the transputer evaluation stack.

The arithmetic instructions take their operands from the top of the evaluation stack (**Areg, Breg**) and push the result value back on the stack in **Areg**.

add	Add
alt	Alt start
altend	Alt end
altwt	Alt wait
and	Bit-wise and
bcnt	Byte count

bsub	Byte subscript (**Areg** = **Areg** + **Breg**)
ccnt1	Check count from 1
clrhalterr	Clear halt-on-error
csngl	Check single
csub0	Check subscript from 0
cword	Check word
diff	Difference
disc	Disable channel
diss	Disable skip
dist	Disable timer
div	Divide
enbc	Enable channel
enbs	Enable skip
enbt	Enable timer
endp	End process
fmul	Fractional multiply (32-bit processors only)
gajw	General adjust workspace
gcall	General call (swap **Areg**↔**Iptr**)
gt	Greater than (result 'true' or 'false', placed in **Areg**)
in	Input message
ladd	Long add
lb	Load byte
ldiff	Long difference
ldiv	Long divide
ldpi	Load pointer to instruction (**Areg** is byte displacement from **Iptr**)
ldpri	Load current priority
ldtimer	Load timer
lend	Loop end
lmul	Long multiply
lshl	Long shift left
lshr	Long shift right
lsub	Long subtract
lsum	Long sum
mint	Minimum integer
move	Move block of memory (src: **Creg** dest: **Breg** len: **Areg**)
mul	Multiply

norm	Normalize
not	Bit-wise not
or	Bit-wise inclusive or
out	Output message
outbyte	Output byte
outword	Output word
prod	Product
rem	Remainder
resetch	Reset channel
ret	Return
rev	Reverse top two stack elements
runp	Run process
saveh	Save high priority queue registers
savel	Save low priority queue registers
sb	Store byte
seterr	Set error
sethalterr	Set halt-on-error
shl	Shift left
shr	Shift right
startp	Start process
sthb	Store high priority back pointer
sthf	Store high priority front pointer
stlb	Store low priority back pointer
stlf	Store high priority back pointer
stoperr	Stop on error
stopp	Stop process
sttimer	Store timer
sub	Subtract
sum	Sum
talt	Timer alt start
taltwt	Timer alt wait
testerr	Test error false and clear
testhalterr	Test halt-on-error
testpranal	Test processor analyzing
tin	Timer input
wcnt	Word count

wsub	Word subscript (**Areg** = **Areg** + 4***Breg**)
xdble	Extend to double
xor	Bit-wise exclusive or
xword	Extend to word

C.4 Additional instructions for T400, T414, T425 and TB

The indirect instructions in this section may only be executed on a T400, T414 or T425 processor.

cflerr	Check single-length floating-point infinity or not-a-number
ldinf	Load single-length infinity
postnormsn	Post-normalize correction of single-length floating-point number
roundsn	Round single-length floating-point number
unpacksn	Unpack single-length floating-point number

C.5 Additional instructions for IMS T800, T801 and T805

The instructions in this section may only be executed on T800, T801 and T805 processors.

C.5.1 Floating-point instructions

The indirect instructions in this section provide access to the T8 series built-in floating-point processor. Note that the instructions beginning with '*fpu* . . .' are doubly indirect: they are accessed by loading an *entry code* constant with a *ldc* instruction, then executing an *fpentry* instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the *fpentry* instruction is also available.

The floating-point load and store instructions use the *integer* **Areg** as a pointer to the operand location.

fpadd	Floating-point add
fpb32tor64	Convert 32-bit unsigned integer to 64-bit real
fpchkerr	Check floating error
fpdiv	Floating-point divide
fpdup	Floating duplicate
fpentry	Floating-point unit entry: used to synthesize the '*fpu* . . .' instructions.
fpeq	Floating-point equality
fpgt	Floating-point greater than

fpi32tor32	Convert 32-bit integer to 32-bit real
fpi32tor64	Convert 32-bit integer to 64-bit real
fpint	Round to floating integer
fpldnladddb	Floating load non-local and add double
fpldnladdsn	Floating load non-local and add single
fpldnldb	Floating load non-local double
fpldnldbi	Floating load non-local indexed double
fpldnlmuldb	Floating load non-local and multiply double
fpldnlmulsn	Floating load non-local and multiply single
fpldnlsn	Floating load non-local single
fpldnlsni	Floating load non-local indexed single
fpldzerodb	Floating load zero double
fpldzerosn	Load zero single
fpmul	Floating-point multiply
fpnan	Floating-point not-a-number
fpnotfinite	Floating-point finite
fpordered	Floating-point orderability
fpremfirst	Floating-point remainder first step
fpremstep	Floating-point remainder iteration step
fprev	Floating reverse
fprtoi32	Convert floating to 32-bit integer
fpstnldb	Floating store non-local double
fpstnli32	Store non-local int32
fpstnlsn	Floating store non-local single
fpsub	Floating-point subtract
fptesterr	Test floating error false and clear
fpuabs	Floating-point absolute
fpuchki32	Check in range of 32-bit integer
fpuchki64	Check in range of 64-bit integer
fpuclrerr	Clear floating error
fpudivby2	Divide by 2.0
fpuexpdec32	Divide by 2^{32}
fpuexpinc32	Multiply by 2^{32}
fpumulby2	Multiply by 2.0
fpunoround	Convert 64-bit real to 32-bit real without rounding
fpur32tor64	Convert single to double

fpur64tor32	Convert double to single
fpurm	Set rounding mode to round minus
fpurn	Set rounding mode to round nearest
fpurp	Set rounding mode to round positive
fpurz	Set rounding mode to round zero
fpuseterr	Set floating error
fpusqrtfirst	Floating-point square root first step
fpusqrtlast	Floating-point square root end
fpusqrtstep	Floating-point square root step

C.6 Additional instructions for IMS T225, T400, T425, T800, T801, T805

The indirect instructions in this section supplement the T414's integer instruction set.

bitcnt	Count the number of bits set in a word
bitrevnbits	Reverse bottom n bits in a word
bitrevword	Reverse bits in a word
crcbyte	Calculate CRC on byte
crcword	Calculate Cyclic Redundancy Check (CRC) on word
dup	Duplicate top of stack
pop	Pop processor stack
lddevid	Load device identity
wsubdb	Form double-word subscript

The following 2-dimensional block move instructions apply to the **IMS T400, T425, T800, T801 and T805** only:

move2dall	2-dimensional block copy
move2dinit	Initialize data for 2-dimensional block move
move2dnonzero	2-dimensional block copy non-zero bytes
move2dzero	2-dimensional block copy zero bytes

C.7 Additional instructions for the IMS T225, T400, T425, T801 and T805

The indirect instructions listed in this section provide debugging and general support functions.

clrj0break	Clear jump 0 break enable flag
setj0break	Set jump 0 break enable flag

testj0break	Test if jump 0 break flag is set
timerdisableh	Disable high priority timer interrupt
timerdisablel	Disable low priority timer interrupt
timerenableh	Enable high priority timer interrupt
timerenablel	Enable low priority timer interrupt
ldmemstartval	Load value of **MemStart** address

C.8 Instructions supported by 'sequential code insertion'

The instructions in this section can be used when 'sequential code insertion' is enabled by the 'G' compiler option. **Note**: Only use those instructions which exist on the target processor may be used. For example, floating point instructions (those beginning with *fp*) may not be used on T4 series transputers.

adc	*add*	*and*	*bcnt*	*bitcnt*
bitrevnbits	*bitrevword*	*bsub*	*ccnt1*	*cflerr*
cj	*crcbyte*	*crcword*	*csngl*	*csub0*
cword	*diff*	*div*	*dup*	*eqc*
fmul	*fpadd*	*fpb32tor64*	*fpchkerr*	*fpdiv*
fpdup	*fpeq*	*fpgt*	*fpi32tor32*	*fpi32tor64*
fpint	*fpldnladddb*	*fpldnladdsn*	*fpldnldb*	*fpldnldbi*
fpldnlmuldb	*fpldnlmulsn*	*fpldnlsn*	*fpldnlsni*	*fpldzerodb*
fpldzerosn	*fpmul*	*fpnan*	*fpnotfinite*	*fpordered*
fpremfirst	*fpremstep*	*fprev*	*fprtoi32*	*fpstnldb*
fpstnli32	*fpstnlsn*	*fpsub*	*fptesterr*	*fpuabs*
fpuchki32	*fpuchki64*	*fpuclrerr*	*fpudivby2*	*fpuexpdec32*
fpuexpinc32	*fpumulby2*	*fpunoround*	*fpur32tor64*	*fpur64tor32*
fpurm	*fpurn*	*fpurp*	*fpurz*	*fpuseterr*
fpusqrtfirst	*fpusqrtlast*	*fpusqrtstep*	*gt*	*j*
ladd	*lb*	*ldc*	*lddevid*	*ldiff*
ldinf	*ldiv*	*ldl*	*ldlp*	*ldmemstartval*
ldnl	*ldnlp*	*ldpi*	*ldpri*	*ldtimer*
lmul	*lshl*	*lshr*	*lsub*	*lsum*
mint	*move*	*move2dall*	*move2dinit*	*move2dnon-*
zero	*move2dzero*	*mul*	*norm*	*not*
or	*pop*	*postnormsn*	*prod*	*rem*
rev	*roundsn*	*sb*	*seterr*	*shl*
shr	*stl*	*stnl*	*sttimer*	*sub*
sum	*testerr*	*testhalterr*	*testpranal*	*unpacksn*
wcnt	*wsub*	*wsubdb*	*xdble*	*xor*
xword				

ASM pseudo-operations are also permitted when sequential code insertion is enabled.

D Transputer code insertion

This appendix describes the facilities for inserting transputer instructions into occam programs, using the **ASM** construct.

D.1 Inline transputer code insertion

The occam compiler supports the insertion of transputer code directly into an occam program. The facility must be specifically enabled on the command line. Two levels of insertion are available.

D.1.1 Sequential code insertion

'Sequential code insertion' allows access to all transputer instructions on the processor except those which affect parallel processes and scheduling. A list of instructions supported by this facility can be found in section C.8.

D.1.2 Full code insertion

'Full code insertion' allows access to all transputer instructions supported by the processor where the process is running. A list of instructions can be found in Appendix C.

D.2 ASM construct

The **ASM** construct provides the ability to insert transputer code sequences into occam programs.

D.2.1 Syntax

```
process        =     asm.construct

asm.construct  =     ASM
                       { asm.line }

asm.line       =     primary.op constant.expression
                  |  load.or.store.op name
                  |  branch.op :label
                  |  branch.op name
                  |  secondary.op
                  | pseudo.op
                  | labeldef

labeldef      =     :label

primary.op    =     any primary instruction (in upper-case letters)

load.or.store.op =  LDL | LDNL | LDLP | LDNLP
                  | STL | STNL

branch.op     =     J | CJ | CALL

secondary.op  =     any secondary instruction (in upper-case letters)

pseudo.op     =     LD asm.exp
                  | LDAB asm.exp, asm.exp
                  | LDABC asm.exp, asm.exp, asm.exp
                  | ST element
                  | STAB element, element
                  | STABC element, element, element
                  | BYTE {, constant.expression }
                  | WORD {, constant.expression }
                  | ALIGN
                  | LDLABELDIFF :label - :label
                  | RESERVELOWWS constant.expression

asm.exp       =     ADDRESSOF element
                  | ADDRESSOF routine.name
                  | expression
```

Note: Instructions should be specified in uppercase.

ASM instructions

The primary instructions which perform loads and stores are allowed to take a symbolic name as their operand; they evaluate to the primary instruction with an operand *equal to that symbol's offset in workspace*. Note that this means, for example, that `LDL x` where `x` is a non-`VAL` parameter, will return the pointer to `x`. This also means that if `x` is a non-local variable, the operand used will be the variable's offset in the non-local workspace. Primary instructions with symbolic name operands should only be used in special cases; you would normally use the pseudo ops as described below.

The assembler will optimize away primary instructions which are known to be no-ops. These are:

```
AJW 0        ADC 0        LDNLP 0
```

PFIX 0 should be used where a NOP byte is required, or the BYTE pseudo-op could be used.

Secondary instructions, and the *fpentry* instructions, simply expand out to the correct byte sequence, as expected.

Branching to a label defined within the same procedure or function is permitted. (Two labels with the same name may not appear in the same procedure.)

Branching to a PROC or FUNCTION which is in scope is permitted, but it is the responsibility of the programmer to load the parameters for the call correctly.

Pseudo operations

The *pseudo.op* operations are defined as follows:

LD	Loads a value into the **Areg**. May use other stack slots and/or temporaries.
LDAB	Loads values into the **Areg** and **Breg**. The left hand expression ends up in **Areg**. May use other stack slots and/or temporaries.
LDABC	Loads values into the **Areg**, **Breg** and **Creg**. The leftmost expression ends up in **Areg**. May use temporaries.
ST	Stores the value from the **Areg**. May use other stack slots and/or temporaries.
STAB	Stores values from the **Areg** and **Breg**. The left hand element receives **Areg**. May use other stack slots and/or temporaries.
STABC	Stores values into the **Areg**, **Breg** and **Creg**. The leftmost element receives **Areg**. May use temporaries.
BYTE	Inserts the following constant BYTE value(s) into the code. The expression may be either a single BYTE, or a BYTE table or string, or a comma separated list of such items.
WORD	Inserts the following constant INT value(s) into the code. The expression may be either a single integer, or an integer table, or a comma separated list of such items.
LDLABELDIFF	Calculates the difference, *n*, between two labels and inserts a LDC *n*.
RESERVELOWWS	Reserves 'below wokspace' slots. This ensures that the specified number of words are reserved below the current process's workspace, and will not be allocated to any other concurrent process. The specified expression must be a compile-time integer constant.
ALIGN	Inserts zero or more PFIX 0 instructions until aligned to a word boundary. *Currently not implemented.*

Note: that arbitrarily complicated expressions are permitted, including function calls. For example:

```
ASM
  LDABC a[x], y+27, f(p,q,r)
  STABC a[f(w,x,y)], z, a[9]
```

Expressions used in *load* pseudo-ops must be word sized or smaller. To load a floating point value, use a `LD ADDRESSOF` *name* to load its address, then a `FPLDNLSN` or `FPLDNLDB` as required. Elements used in *store* pseudo-ops must be word sized or smaller.

`ADDRESSOF` operator

The `ADDRESSOF` operator is used in the `LD`, `LDAB`, and `LDABC` pseudo-ops, and can be applied to any variable, constant expression, or routine name. It returns a word containing the machine address of that object.

Special names

The following special names are available as constants inside `ASM` expressions.

`.WSSIZE`	Evaluates to the size of the current procedure's workspace. This will be the workspace offset of the return address, except within a replicated `PAR`, where it will be the size of that replication's workspace requirement.
`.VSPTR`	Evaluates to the workspace offset of the vectorspace pointer. If inside a replicated `PAR`, it points to the vectorspace pointer for that branch only. A compile time error is generated if there is no vectorspace pointer because no vectors have been created.
`.STATIC`	Evaluates to the workspace offset of the static link. If inside a replicated `PAR`, it points to the static link for that branch only. A compile time error is generated if there is no static link.

For example, to determine the return address of a procedure, you would use: `LDL .WSSIZE` There is no checking of 'suitability', hence, for example, `J .WSSIZE` is legal.

Channels

Channels may be accessed in `ASM`; they are considered to be a pointer to a channel word. Thus 'loading' a channel will load a pointer to the channel word, and loading the 'address' of a channel will load a pointer to a pointer to the channel word.

E Glossary

Alias

If two or more expressions denote the same memory address, then the expressions are aliases or one another.

Alias check

A program compilation check that ensures that names are unique within a given scope.

Analyse

A transputer input pin which forces the processor to halt at the next de-scheduling point, to allow the state of the processor to be read. To assert the **Analyse** input on a transputer. In the context of 'analyzing a network', to analyze all transputers in that transputer network.

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.

Big endian

The opposite of **little endian** — see below.

Bootable code

Self-starting program code that can be loaded onto a transputer or transputer network down a **user link** and run. Bootable code is produced by `icollect` from linked units (single transputer programs) or configuration binary files (for configured programs).

Bootstrap

A transputer program, loaded from ROM or over a link after the transputer has been reset or analyzed, which initializes the processor and loads a program for execution (which may be another loader).

Capability

A text string which identifies a transputer **resource** (or resources).

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 a program with a set of physical resources. Used in this manual to refer to the specific case of allocating software processes to processors in a network, and channels to links between processors. The term is also used, depending on the context, to describe the act of deciding on these allocations for a program, the configuration code which describes such a set of allocations, and the act of applying the configurer to a configuration description.

Configurer

The tool which assigns processes and channels on a specified configuration of transputers. The output from the tool is a configuration binary file for input to `icollect`.

Connection manager

The functionality provided by the **Linkops** part of the host file server. Provides and maintains connections to transputer systems across a network and is used by the **session manager** to select a transputer system and maintain unique access to that system.

Core dump

A memory dump. Core dumps are required as part of the process of debugging multitransputer programs that incorporate the root transputer.

Communicating Sequential Processes (CSP)

A theory and notation, developed by Professor Tony Hoare, for describing systems made up of concurrent processes which communicate via channels. The occam model of concurrency is based on CSP.

Deadlock

A state in which one or more concurrent processes can no longer proceed because of a communication interdependency.

Error modes

The compilation mode of a program that determines what happens when a program error (such as an array bounds violation) occurs. Programs are compiled by `icc` in UNIVERSAL mode, which is the mode that can be mixed with HALT and STOP code generated by other INMOS compilers.

Error signal (or error flag)

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 transputer networks. Error signals can be OR-ed together on transputer boards to indicate that an error has occurred in one of the transputers in a network.

Ethernet

A LAN technology based on a passive coaxial cable which transmits at 10 Mbps.

Extended data types

The occam data types **INT16**, **INT32**, **INT64**, **REAL32** and **REAL64**.

External memory interface (EMI)

The signals which connect a transputer to external memory, consisting of address and data buses and a number of control signals. Most of the 32 bit transputers (T4xx and T8xx) have a programmable EMI which can be configured for different types and speeds of external memory device.

Event

An input signal to the transputer which can be used an external interrupt. The event input can be treated by a process as a (zero length) communication.

Free variables

Variables which are referred to in a function or procedure, but declared outside of it.

Gateway

A dedicated computer that connects two or more networks, and routes messages between them.

Hard channels

Channels which are mapped onto **links** between processors in a transputer network (see also **soft channels**).

Host

The computer to which a transputer system is connected and which possibly also provides file system access and terminal I/O.

Host file server

A **server** which provides access to the filing system and terminal I/O of a host operating system.

Include file

A file containing source code which is incorporated into a program using the C `#include` (`#INCLUDE` for occam) directive. Include files are, by convention, given the `.h` extension in C; occam include files are given the extension `.inc`.

LAN (Local Area Network)

Any computer network that works over short distances at high speeds.

Library

A collection of separately compiled procedures or functions, created by the toolset librarian `ilibr`, which may be shared between parts of a program or between different programs.

Library build file

A file containing a list of input files for the librarian tool `ilibr`. Each file forms a separately loadable module in the library. Library build files should have the `.lbb` extension.

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.

In the context of program compilation, collecting together all the compiled code for a program, resolving all references and placing the collected code into a single file.

Linker

The program or tool which links a program or compilation unit.

Linkops

The recommended INMOS link interface, used by `iserver` 1.5.

Little endian

The transputer is totally 'little endian', i.e. less significant data is always held in lower addresses. This applies to bits in bytes, bytes in words and words in memory.

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

Depending on context may refer to a conventional computer network or a set of interconnected transputers.

Object code

Intermediate code between source and **bootable code**. Object code cannot be directly loaded onto a transputer and run. The compiler and **linker** tools generate object code.

Peek and poke

To read (peek) and write (poke) locations in a transputer's memory via a **link**, while the transputer is waiting to be booted.

PostScript

PostScript is a device-independent, interpreted language for describing the layout of text and graphics on a page. It is used by a large number of printers and software applications as the standard means of transferring graphics data.

Preamble

The part of a transputer **loader** program that initializes the state of the processor.

Priority

In the transputer, the priority level at which the currently executing process is being run. INMOS transputers support two levels of priority, known as 'high' and 'low'.

Process

Self-contained, independently executable code.

Protocol

The pattern (type, etc.) of communications between two processes, often including communications on more than one channel. Protocols can be defined in occam and must be specified when a channel is declared.

Reset

The transputer system initialization control signal.

Root transputer (or root processor)

The processor in a transputer network which is physically connected to the host computer, and through which the transputer network is loaded.

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 re-compiled (see also **makefile**).

Server

A program running on a host computer which provides access to the filing system and terminal I/O of the host for the transputers, or access to the transputer system from the host. The server can also be used to load the program onto the network.

Session manager

That part of the **server** which maintains unique access (a session) to a transputer system when requested by a user.

Soft channels

Channels declared and used within a process running on a single transputer (see also **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 one board to control another board connected to 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.

Transputer Module (TRAM)

A range of small printed circuit boards which typically hold a transputer, some memory and, optionally, some other application specific hardware. TRAMs can be interconnected via links to build systems based on a number of motherboard architectures. For more information see the *i*q systems databook.

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.

User link

The connection of a transputer **resource** to a **host** computer.

Vector space

The data space required for the storage of arrays within occam code (see also **workspace**).

Worm

A program that distributes itself through a network of transputers (perhaps with an unknown topology) and allows all the processors in the network to be loaded, tested or analyzed.

Workspace

The data space required by an occam process. When used in contrast to **vector space**, refers to the data space required for *scalars* within the occam code.

F Bibliography

F.1 Transputers

The transputer databook (Third Edition 1992)

INMOS Ltd, July 1992
INMOS document number 72 TRN 203 02

The military and space transputer databook (First Edition 1990)

INMOS Ltd, July 1990
INMOS document number 72 TRN 224 00

Transputer instruction set: A compiler writer's guide

INMOS Ltd
Prentice Hall 1988

Transputer Hardware and systems design

JC Hinton and AL Pinder
Prentice Hall 1993

The transputer handbook

Ian Graham and Tim King
Prentice Hall 1990

F.2 C programming

The C programming language (First Edition)

Brian W Kernighan & Dennis M Ritchie
Prentice Hall 1978

The C programming language (Second Edition — ANSI C)

Brian W Kernighan and Dennis M Ritchie
Prentice Hall 1988

C: A reference manual (Second Edition — ANSI C)

Samuel P Harbison and Guy L Steele
Prentice Hall 1987

American National Standard for Information Systems – Programming Language C

American National Standards Institute 1990
Ref. Doc. X3J11/88-159

F.3 occam programming

occam 2 reference manual

INMOS Ltd
Prentice Hall 1988

A tutorial introduction to occam programming

D Pountain and D May
Blackwell Scientific 1987

An introduction to occam 2 programming

KC Bowler, RD Kenway, GS Pawley and D Roweth
Chartwell-Bratt 1987

Programming in occam 2

A Burns
Addison-Wesley 1988

occam 2

A Gallently
Piman 1989

Programming in occam *2*

G Jones and M Goldsmith
Prentice Hall 1988

Concurrent programming in occam *2*

J Wexler
Ellis Horwood 1989

F.4 INMOS technical notes

The transputer applications notebook:
Architecture and software (First Edition 1989)

INMOS Ltd, May 1989
INMOS document number 72 TRN 206 00

The transputer applications notebook:
Systems and performance (First Edition 1989)

INMOS Ltd, June 1989
INMOS document number 72 TRN 205 00

IMS B004 IBM PC add-in board

Technical note 11
INMOS document number 72 TCH 011

Notes on graphics support and performance improvements on the IMS T800

Technical note 26
INMOS document number 72 TCH 026

Security aspects of occam *2*

Technical note 33
INMOS document number 72 TCH 033

Simple real-time programming with the transputer

Technical note 51
INMOS document number 72 TCH 051

Using the occam toolsets with non-occam applications

Technical note 55
INMOS document number 72 TCH 055

F.5 Development systems

The transputer development and iq systems databook (Second Edition 1991)

INMOS Ltd, 1991
INMOS Document Number 72 TRN 219 01

IMS B300 TCPlink hardware manual

INMOS Limited, June 1991
INMOS Document Number 72 TRN 229 01

ＡＮＳＩ Ｃツールセット・ユーザー・マニュアル（日本語版）
（Ｄ４２１４Ｂ、Ｄ５２１４Ｂ、Ｄ７２１４Ｃバージョン）

翻訳：　新日本製鉄（株）エレクトロニクス研究所電子システム研究部
　　　　貝塚洋、田内宏明、斉藤知人、金子博昭、青柳雄大
　　　ＳＧＳトムソン・マイクロエレクトロニクス（株）
　　　　梅本剛
発行所：　日刊工業新聞社（株）

F.6 References

Software manual for the elementary functions

WJ Cody and WM Waite
Prentice Hall 1980

The art of computer programming
2nd edition, Volume 2: *Seminumerical algorithms*

DE Knuth
Addison Wesley 1981

IEEE Standard for binary floating-point arithmetic

ANSI-IEEE Std 754-1985

Communicating sequential processes

CAR Hoare
Prentice Hall

Index