occam 2.1 Toolset
User Guide

SGS-THOMSON
MICROELECTRONICS

August 1995

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures)
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines).
- P7250 TMP (Transputer Macrocell Project).
- P7267 OMI/STANDARDS.

Document Number: 72 TDS 366 02

Contents overview

Basics

Advanced techniques

Appendices

Contents

Preface

Host versions

The documentation set which accompanies the occam 2.1 toolset is designed to cover all host versions of the toolset:

- IBM 386 PC compatible running MS-DOS
- Sun 4 systems running SunOS or Solaris.

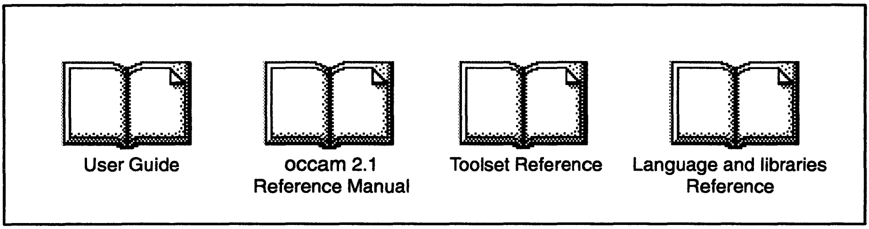
About this manual

This manual is the *User Guide* to the occam 2.1 toolset.

The manual describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two parts; *Basics* which describes each of the main stages of the development process, how to configure occam programs for the transputer and how to build and load them. It also includes a *Getting started* tutorial. The *Advanced Techniques* part is aimed at more experienced users. The *Appendices* contain some reference material, a glossary of terms and a bibliography. Some chapters are in common with other toolsets e.g. *Developing programs for the transputer* and *Mixed language programming.*

About the toolset documentation set

The toolset documentation set comprises the following volumes:



- *occam 2.1 Toolset User Guide*

  This manual.

- *occam 2.1 Reference Manual*

  Provides a description and definition of the occam 2.1 programming language, including the changes from occam 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 several toolset products e.g. the occam 2.1 toolset and the ANSI C toolset and the documentation reflects this. The appendices provide details of toolset conventions, transputer types, memory configuration files and the configuration languages.

- *occam 2.1 Language and Libraries Reference Manual*

  Provides a language reference for the toolset and implementation data. A list of the library routines provided is followed by detailed information about each function and procedure.

Other documents

Other documents provided with your toolset product include:

- *Delivery manual*

  This document gives installation data and is host specific.

Transputer targets supported by this toolset

The first generation of IMS T2xx, T4xx and T8xx transputers are referred to as the 'T2/T4/T8-series'. The new product family based around the IMS T9000 transputer is referred to as the 'T9-series'. The ST20450 may be called an ST20 or the IMS T450 depending on the context, and is included in the T2/T4/T8-series family unless otherwise stated.

The documentation set provided with this toolset tends to be generic, e.g. it covers different transputer types and in some cases different programming languages: C and occam. The occam 2.1 toolset supports occam 2.1 running on T2/T4/T8-series and ST20450 transputers. References in the documentation to other languages or transputer targets should be ignored.

Documentation conventions

The following typographical and notational 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.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Braces { }	Used to denote optional items in a command syntax which may be repeated.
Ellipsis ...	Used to denote a range or 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.

Part 1: 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 transputer 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 used singly or connected one-to-another by their serial links in application-specific ways. They may be used as powerful dedicated microprocessors or as building blocks for complex parallel processing networks.

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

- 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.
- Real time clocks.
- On the T8 series, an integral floating point unit.

Figure 1.1 shows the generalized architecture of a transputer with four links.

1.1.1 Transputer links

Links are high performance communications interfaces. They 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 T400 transputer which has 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 links from a ROM on a single transputer. Both these models are supported by the transputer toolsets. For ease of development, stand-alone systems and software are generally developed using a hosted system and then the software is put into a ROM as a final development stage.

T2/T4/T8-series transputer links consist of two wires, one in each direction, and uses an asynchronous bit-serial (byte-stream) protocol. Each bit received is sampled five times and hence the links are referred to as over-sampled links (OS-Links). Each link provides a pair of channels, one in each direction and can operate at up to 20 Mbits/sec, providing a bidirectional bandwidth of 2.4 Mbytes/sec.

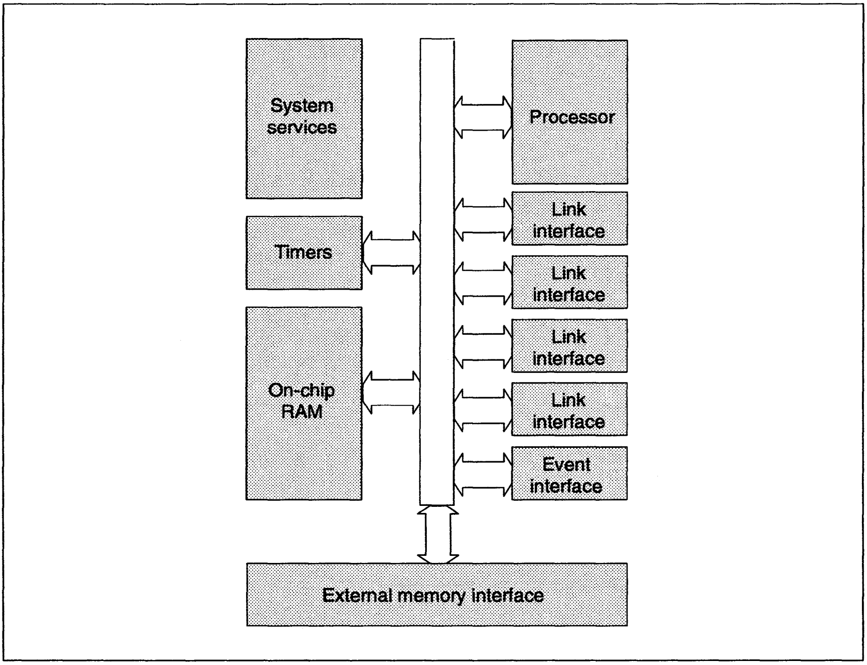


Figure 1.1 Transputer architecture

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 *channels*. Processes waiting for input or output or waiting for a processor to become available, consume no CPU resources, and process context switching time is less than one microsecond. This means that most applications require no operating system, with a corresponding improvement in performance and cost savings.

The scheduling mechanism may also be used for performing interrupts, which makes interrupt-handling simple and reliable.

1.1.3 Real time programming

The transputer has features which provide direct hardware support for real-time programming. The key features are:

- Direct and efficient implementation of parallel processes in hardware.
- Two priority levels for 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 and PORTs at specific addresses in memory, for accessing memory mapped devices.

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

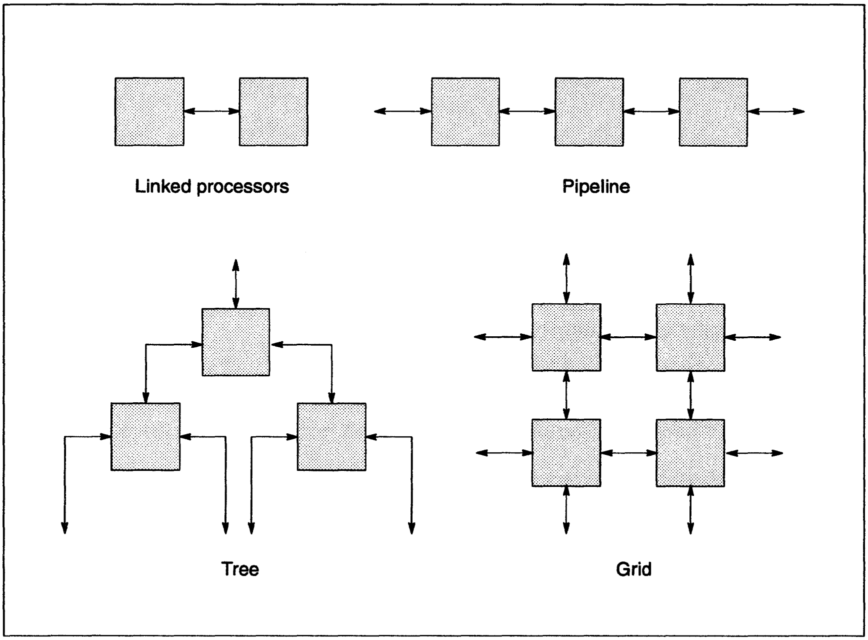


Figure 1.2 Transputer networks

1.1.4 Multi-transputer systems

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

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

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, either to simplify the software or to make the software portable onto multi-processor systems.

Parallelism can be designed into a program at two levels by dividing the program into a number of independent communicating sequential processes capable of operating in parallel. Such processes can either be run on a single transputer, using the transputer's hardware scheduler, 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. Such programs can exploit the transputer architecture and software support provided by the toolsets and systems products, as described in 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.* A channel is a one-way, point-to-point communication path that allows processes to exchange data and synchronize their activity. Further details can be found in *Communicating Sequential Processes* by 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 methodologies.

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, link switches and link adaptors for interfacing to other processors and devices.

A wide range of SGS-THOMSON TRAnsputer Modules (TRAMs) and motherboards is available for specific hosts. A TRAM is a transputer with an amount of memory and possibly some other application specific hardware, mounted on a circuit board. TRAMs can be interconnected via links to build systems based on a number of motherboard architectures. SGS-THOMSON transputer board products 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 transputer toolsets are complete software 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 sequential, parallel and embedded software applications.

2 Overview of the toolset

This chapter introduces the occam 2.1 toolset. It describes the main features of the toolset and provides introductions to the occam 2.1 compiler, the toolset linker, the configuration system, and mixed language programming. A summary of the toolset components is given at the end of this chapter.

2.1 Introduction

The occam 2.1 toolset is a software cross-development system for transputers, hosted on a 386 PC with MS-DOS or a Sun 4 system with SunOS. It consists of an occam 2.1 compiler, a multi-language linker, configuration and code collection tools and program development tools.

The program development tools include a librarian, an object code lister, a makefile generator, memory interface configurers 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.

The toolset is supplied with INQUEST and host interface software. INQUEST includes an interactive and post-mortem debugger and various profiling tools. The INQUEST tools are described in the *INQUEST User and Reference Manual.* The host interface software includes an application loader and a host file server.

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 2.1 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 manually 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.
- Support for ST20450 (T450) targets, featuring the following:

- byte and half word instructions used to support 8-bit and 16-bit integer operations;
- PORTs implemented using the device access instructions on ST20450 processors;
- memory configuration.

2.3 Standard object file format

The current range of transputer 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 2.1 compiler

The occam 2.1 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 2.1 which is described in the *occam 2.1 Reference Manual.* The compiler implements a number of non-standard language extensions, as described in Appendix A in the *occam 2.1 Toolset Language and Libraries Reference Manual.*

The compiler targets the current range of T2/T4/T8-series transputers and the ST20450. Code may be generated for specific processor types or for related groups, as described in Appendix B in the *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 object files which are to be linked together, 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 2.1 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 2.1 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 3.13 of the *Toolset Reference Manual.*

Other language extensions supported by the compiler are:

- assembly code insertion;
- memory placement;
- extended channel retyping.

See Appendix A of the *occam 2.1 Toolset Language and Libraries Reference Manual* for details.

2.4.3 occam libraries

A comprehensive set of libraries and include files is 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 Chapter 2 in the *occam 2.1 Toolset Language and Libraries Reference Manual*). The compiler automatically loads the correct set of routines for the selected error mode and transputer type. Compiler libraries are specified to the linker by means of target-specific *linker indirect files*; see section 4.3.1.

The user libraries provide application-level support. There are libraries to support:

- single length, double length, and T4-optimized maths;
- file-based and stream-based i/o;
- string handling;
- type conversion;
- link error handling;
- CRC coding;
- debugging.

Constants and definitions are supplied in include files. See the *occam 2.1 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 is provided which enables a separately compiled and linked occam procedure to be loaded and called at run-time and incorporated in a running occam program. This facility is aimed at experienced toolset users.

Full descriptions of these facilities are given in Chapter 12.

2.5 Multi-language linker

The toolset linker takes compiled code and libraries and generates a linked unit in TCOFF format. Input code can be generated by any compiler which generates TCOFF code, for example, the 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 4.3.1.

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 10.4 of the *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 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. Generally, existing configuration code can be reused – virtual routing will be employed where required unless virtual routing is specifically disabled by the configurer NV option.

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 the `IMS_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, excluding the INQUEST tools, are summarized in table 2.1. Descriptions of the tools can be found in chapter 4 which also describes the main stages of program development.

2.8 Toolset summary

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.
`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.
`ilaunch`	The Windows launch tool. Used for setting Windows environment parameters.
`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.
`imem450`	The memory configuration tool for T450 targets. Used for evaluating and defining the memory configuration.
`irun`	The application loader. Used for loading built programs onto the target.
`iset`	The Windows parameter tool. Used for setting Windows environment parameters.
`oc`	The occam 2.1 compiler. Compiles code for the current range of transputers.
`occonf`	The occam configurer. Reads a configuration description and produces a configuration data file for the code collector.

Table 2.1 The occam 2.1 toolset

3 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. A detailed description of program development is given in chapter 4. chapter 7 provides examples of multi-transputer programming.

3.1 Introduction

In order to create and run a transputer executable file, the following steps must be followed:

1 The source files are compiled with the occam 2.1 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 collected to create a bootable program. 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. The INQUEST tools do not support unconfigured programs, and there are many other advantages to configuring which will become apparent as the procedures are described.

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

3.2 Running the examples

In the following examples, the programs are compiled and executed on a single IMS T425 with 1Mbyte of memory available. If you have a different model of 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 *Toolset Reference Manual.*

The examples assume the existence of

- an environment variable `TRANSPUTER` which defines the name of a *connection* to a target network on which to load the program, and

- an *AServer database* file to define that connection.

See the *Delivery Manual* which accompanies this toolset and the `irun` documentation (chapter 15 of the *Toolset Reference Manual*) for more details.

The examples also assume the existence of the environment variable `ISEARCH`, which gives the tools a search path to find libraries and include files which are not in the current directory. 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.

3.2.1 Sources

Source files for the example used in this chapter are supplied with the toolset under the `examples` directory.

3.3 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 is in the file `simple.pgm`.

The program is listed below with line numbers.

```
1.  #INCLUDE "hostio.inc" -- contains SP protocol
2.  PROC simple (CHAN OF SP fs, ts)
3.    #USE "hostio.lib"
4.    [1000]BYTE buffer :
5.    BYTE result:
6.    INT length:
7.    SEQ
8.      so.write.string    (fs, ts,
                                "Please type your name :")
9.      so.read.echo.line  (fs, ts, length, buffer, result)
10.     so.write.nl        (fs, ts)
11.     so.write.string    (fs, ts, "Hello ")
12.     so.write.string.nl (fs, ts,
                                 [buffer FROM 0 FOR length])
13.     so.exit            (fs, ts, sps.success)
14. :
```

Line 1 in the program includes the file `hostio.inc`. This file contains the definition of the 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. This line must be the first line because line 2 refers to the `SP` protocol defined in `hostio.inc`.

The procedure `simple` is then declared. All the working code is contained within this procedure. The host i/o library `hostio.lib` is referenced by the `#USE` directive in line 3. This library contains all the procedures used by the program. See chapter 4 in the occam *2.1 Toolset Language and Libraries Reference Manual* for descriptions of these 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 on line 8, reads the name from the keyboard on line 9, and displays a greeting on the screen in lines 11 and 12.

The last statement, on line 13, calls a library procedure which terminates the host file 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 from the keyboard.

3.3.1 Compiling the program

In order to compile the program for an IMS T425 use the following command line:

```
oc simple -t425
```

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 given as an IMS T425; for information about compiling for other transputer types, see section 3.4.

By default, the compiler enables interactive debugging with INQUEST and compiles the program in HALT mode; see chapter 3 of the *Toolset Reference Manual* for a description of the modes.

3.3.2 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
```

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 in the linker indirect file `occama.lnk` which is specified by the `F` option.

The `F` option introduces a *linker indirect file* which is used to link in the correct compiler libraries. For more details see chapter 10 in the *Toolset Reference Manual.* If interactive debugging were required then the debugging libraries should be linked in, as described in the *INQUEST User and Reference Manual.*

It is necessary to specify to the linker what the transputer target is. The toolset can produce code for a range of transputers and the linker must then be told which the actual target will be. In this example the chosen target is an IMS T425.

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.

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, as described in section 3.5.

3.3.3 Configuring the program

In order to configure the program, a description is required of the network it is to 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.

3.3.4 Collecting the program

The final build stage is to collect all the parts of the program with the bootstrap and loading and routing code 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`.

3.3.5 Running the program on a transputer board

To load the bootable file onto a transputer board and run it, use the application loader tool `irun`.

On a Sun, the `irun` command line can be entered at a SunOS prompt. On a PC running Windows, the `irun` command may be entered on the command line of a DOS window if `ilaunch` is running. Alternatively `irun` may be started in the normal Windows manner using the file manager **Run..** command or an icon.

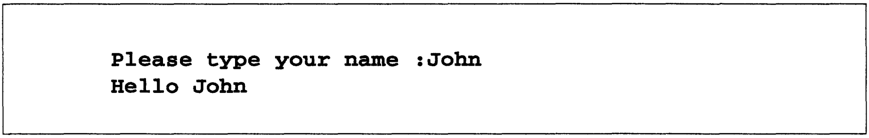
The `irun` command line is:

```
irun simple.btl
```

The connection to the transputer board is taken from the `TRANSPUTER` environment variable.

The command line specifies the file to be booted. The command has the effect of resetting the target network, opening communication between the host and the target, and loading the program onto the target network. For further information about `irun` options see chapter 15 in the *Toolset Reference Manual.*

Figure 3.1 shows an example of the screen display obtained by running `simple.btl` for a user called `John`.



```
Please type your name :John
Hello John
```

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

3.3.6 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 stand-alone systems nor to systems which boot from ROM, and requires the program to be contained within a *single linked unit*. The INQUEST debugger cannot be used to debug programs created in this way. This facility is provided for compatibility with previous versions of the toolset and is not recommended.

Non-configured programs require a fixed procedural interface, because the parameters cannot be defined in the configuration description. Line 2 of `simple.occ` on page 16 would therefore become:

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

The `free.memory` parameter represents the spare memory not allocated by the tools. The total size of the memory available is taken from the environment variable `IBOARDSIZE`.

Unconfigured 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
```

The bootable file `simple3.btl` is created. This can be loaded and run in the same way using `irun`.

3.4 Compiling and linking for other transputer types

If you are using a transputer other than an IMS T425 you should specify the appropriate target transputer type for the compilation and linking operations. Appendix B in the *Toolset Reference Manual* describes the options available. The same processor type must be specified to the compiler, linker, and in configuration description, 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 10 of the *Toolset Reference Manual*).

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

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

Similarly for an ST20450:

```
oc simple -t450
ilink simple.tco hostio.lib -f occam450.lnk -t450
```

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.

3.5 Using the Makefile generator

As an alternative method of program building, 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 12 in the *Toolset Reference Manual* for a full description of the tool.

imakef serves two purposes:

- It enables the user to generate a makefile which can be used to generate a target file automatically (e.g. a bootable file) without having to manually request 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 so that only the affected parts need to be rebuilt.

In order that **imakef** can identify file types, target processor types and error modes, a different system of file extensions must be used to that used in the examples above. See section 12.3 in the *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
```

This directs **imakef** to build a makefile to create a bootable file **simple2.btl**. The **.btl** extension tells **imakef** to refer to a configuration description file **simple2.pgm**. This file can be found under the **examples** directory. Within the **.pgm** file, the correct file extension is used to reference the linked unit for **imakef**. For example:

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

tells **imakef** that the Makefile must compile the program for an IMS T425 in HALT error mode. For other transputer types and error modes use different suffixes, as described in section 12.3 in the *Toolset Reference Manual.*

imakef generates the Makefile **simple2.mak**. To build the program run your **make** program using the Makefile **simple2.mak**:

```
make -f simple2.mak
```

This creates the bootable file **simple2.btl** which can be run in the normal way using **irun**.

4 Developing programs for the transputer

This chapter gives an overview of the program development cycle using the occam 2.1 toolset. 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 gives brief details of environment variables, and host dependencies. Chapter 8 describes loading and running the built application on the target hardware.

Further information about the tools for building and loading code can be found in the *Toolset Reference Manual.* The INQUEST debugging, profiling and analysis tools are described in the *INQUEST User and Reference Manual.*

4.1 Program development using the toolsets

Programs are developed on the user's host system before down-loading for execution on either a single transputer or a network of transputers.

Executable code may be loaded onto a transputer either from ROM or from the host system. Code from the host file system is loaded via a single transputer link from the host to the *root transputer*, i.e. the transputer connected to the host. 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. Figure 4.1 summarizes the main development stages and figure 4.2 illustrates how the tools are used to build a program.

The following sections concentrate on how to develop programs which are loaded from the host system. Any system that is to be loaded from ROM when completed, is normally loaded from the host during development and then converted to loading from ROM as a final development stage. Developing programs for loading from ROM is described briefly in section 4.6 and more fully in chapter 13.

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 using the support provided for multi-tasking and parallel programming.

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

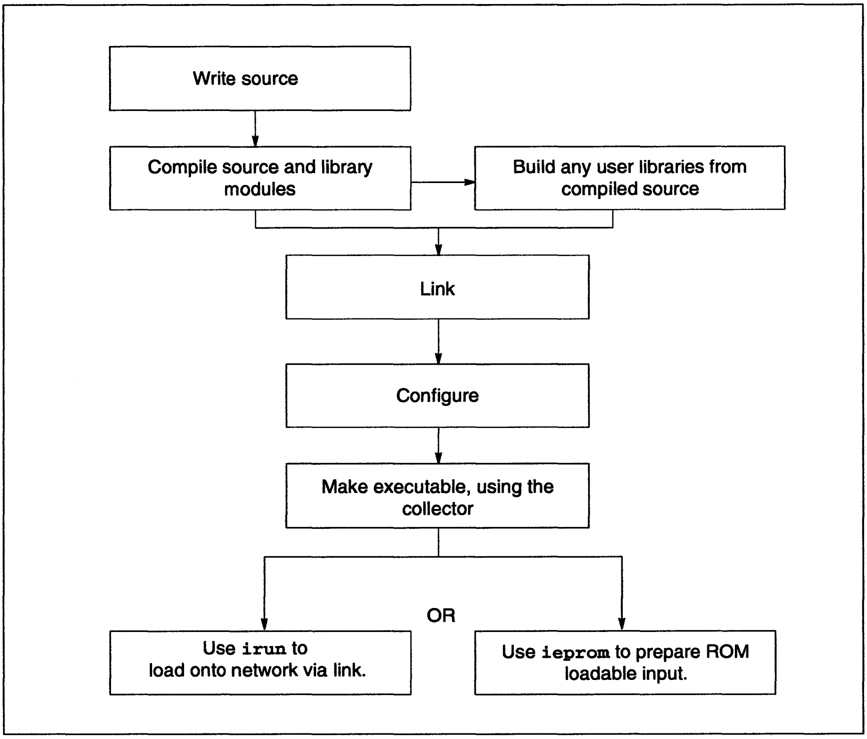

Write source
Compile source and library modules
Build any user libraries from compiled source
Link
Configure
Make executable, using the collector
OR
Use irun to load onto network via link.
Use ieprom to prepare ROM loadable input.

Figure 4.1 Main development stages

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 2.1 language definition.

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 appropriate transputer type or for a transputer class covering several compatible types. (More information about transputer types and classes is given in the Appendix B of the accompanying *Toolset Reference Manual*). Commonly used object code can be combined into libraries using the toolset librarian `ilibr`.

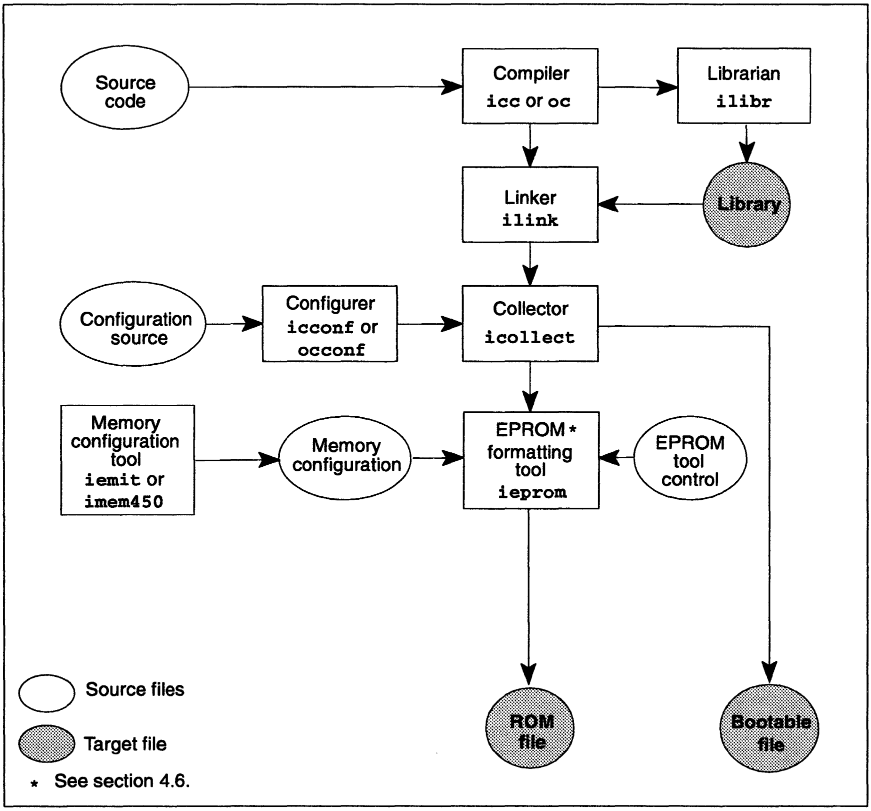


Figure 4.2 Program build model

4 Link the compiled units

The compiled object files and libraries are linked together using the toolset linker `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 one processor of 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 the hardware. It is achieved by writing a *configuration description*, assigning linked units to specific processors and 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 binary file*. Configuration is used for both single and multi-processor transputer programs.

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

6 **Generate an executable file**

Before a program can be run it must be made executable, i.e. *bootable*. This involves adding bootstrap and loading information. The result is a single executable file that it can be directly loaded onto the target. The collector can also include debugging and profiling kernels.

This is achieved using the collector tool `icollect`. The configuration binary file generated by the configurer is read by the collector. The collector can generate either a file which is suitable for loading onto the target hardware from the host via a hardware serial link or one for loading from ROM. The default behavior of the tool is to produce a boot-from-link bootable.

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

7 **Memory configuration**

Some transputers (IMS T400, T414, T425, T800, T805) have a programmable memory interface which may be optionally configured using `iemit`. This tool generates a memory configuration file which can then be loaded into ROM; see section 4.6.

For the ST20450 (T450) transputer a memory configuration file is needed if there is any external memory. The memory configuration code for T450 TRAM modules is held in a ROM, so the memory interface will automatically be configured when the hardware is reset. Memory configuration files for other ST20450 hardware are generated using `imem450`. Generating such files is described in chapter 14.

8 **Load and run the program**

A bootable boot-from-link file is loaded and run on the target hardware down a hardware serial link using the application loader `irun`, as described in chapter 8. Once loaded the code begins to execute immediately. The application loader maintains the environment that supports the program's communication with the host.

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

4.1.1 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 code built in this way cannot be debugged with the INQUEST interactive debugger.

If this mode is used, then the environment variable `IBOARDSIZE` should be set up; see section 4.10.3.

4.2 Compiling

The occam 2.1 compiler produces compiled code for a specific processor type or for a group of related processors called a transputer class. This compiler supports the IMS T212, M212, T222, T225, T400, T414, T425, T800, T801, T805 and ST20450 (also known as the T450). Each compiler has the same set of options to select the target transputer; these are listed in the Appendix B of the *Toolset Reference Manual.* The role of transputer types and classes in compilation and program development is also described in that appendix.

The compiler is capable of compiling code in one of three error modes. 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 HALT mode, and when compiling for this error mode you may omit the error mode option. The error modes are described in section 6.2.1.

The compiler enables interactive debugging by default unless the compiler `D` or `Y` option is used.

The current range of transputer compilers generate code files in a format 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 the Toolset is a set of libraries which provide run-time support, input and output operations, mathematical functions etc. The libraries supplied with this toolset are introduced in chapter 2.

Support is also provided for language extensions, interactive debugging, assembler inserts and software configuration of a network. Other operating features of the compiler may be changed by options and directives; see chapter 3 in the *Toolset Reference Manual.*

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

If the compilation succeeds, the compiler creates a new object file in the current directory. The filename for the new file may be specified on the command line, or the default filename is derived from the name of the source file with the file extension `.tco`.

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.

4.3 Tools for building executable code

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

- `ilink` – the toolset linker which links separately compiled units
- `occonf` – the configurer tool which generates a configuration binary file.
- `icollect` – the code collector which generates an executable 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 file. The executable file contains all the information needed to distribute, load, and run the program on a specific network of transputers.

4.3.1 Linker – `ilink`

The toolset linker `ilink` links separately compiled modules or object files and libraries together, 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.

The default output file is derived from the name of first file listed either on the command line or in a linker indirect file.

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 used for the compilation.

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 3.13.7 and 10.4.6 in the *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.

Linker indirect files

A *linker indirect file* is a text file containing a list of files and commands to the linker. Libraries to be linked in with the program may be listed in a linker indirect file, which is specified on the linker command line using the F option. The usual extension for a linker indirect file is **.lnk**. For occam code, linker indirect files may be generated by **imakef**.

For example, a linker indirect file **hello.lnk**, listing all the files to be linked may be then the following command line will cause the linker to link the listed files for an IMS T425:

```
ilink -f hello.lnk -t5
```

Standard linker indirect files, which specify the compiler libraries, are supplied with the toolset, and may be explicitly specified on the command line or **#included** in a user-written indirect file. The correct standard linker indirect file must be specified, depending the type of the target transputer, as shown in table 4.1. For further details of the compiler libraries see the *occam 2.1 Toolset Language and Libraries Reference Manual.*

Linker indirect file	Target transputers
occam2.lnk	T212, M212, T222, T225
occama.lnk	T400, T414, T425, TA, TB
occam450.lnk	ST20450
occam8.lnk	T800, T801, T805

Table 4.1 Standard occam linker indirect files

Each standard linker indirect 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 the libraries it requires from the list supplied in the indirect file.

For example, the following linker indirect file will link **hello.tco** with the host i/o library and the compiler library:

```
hello.tco
hostio.lib
#include occama.lnk
```

If the above lines are in the file **hello.lnk** then the linker command line

```
ilink -f hello.lnk -t5
```

is equivalent to:

```
   ilink hello.tco hostio.lib -f occama.lnk -t5
```

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 4.1 is always used. When the main program is written in C, one of the standard C start-up files should be used, as described in the *ANSI C Toolset User Guide*. However, if an occam program calls functions written in C then it must be linked with C modules. In this case the standard C start-up files are not suitable and one of the C linker indirect files without a main entry point definition 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.

For further details of mixed language programming, see chapter 11.

4.3.2 Configurer – `occonf`

The configurer generates configuration information for transputer networks from a textual *configuration description*. The tool prepares distribution information about 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 a transputer *configuration language* appropriate to the configurer used. The `occonf` configuration language is described in chapters 7 and 10.

4.3.3 Code collector – `icollect`

Configured code cannot be loaded directly onto a transputer network for two reasons:

1 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.

2 Code to be run on a board which boots from link, such as the IMS B008, requires the addition of bootstrap information to initialize the processor, load the program and start it running.

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

The code collector takes the binary file generated by the configurer (which refers to the linked code) and generates a single file that can be loaded and run on a transputer network. The collector generates all 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 code on each processor.

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

4.4 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 supplied with the toolset and described below. Boot-from-ROM code is processed by the EPROM programming tool and is introduced in section 4.6.

Chapter 8 describes how to load and run application programs in more detail.

4.4.1 Host file server

The AServer is provided with the Toolset, which can act as a combined host server and application loader tool. When invoked to load a program it both loads the code onto the transputer hardware and provides run-time services on the host for the transputer program including screen, keyboard and file i/o.

The version of the AServer supplied will depend on the host and hardware interface supported by the toolset. Chapter 8 describes how to load application programs onto the target hardware. The application loader `irun` is part of the AServer and is described in more detail in chapter 15 of the *Toolset Reference Manual*.

The AServer can be extended and customized by adding extra services, such as graphics servers. With the AServer, this can be done without modifying the standard host file server. Details of how to add services are given in the *AServer Programmers' Guide*.

4.4.2 Skip bootables

The skip bootables allow an application to run on a transputer subnetwork which is not directly connected to the host, but is connected via one or more intermediate transputers. For example, the root transputer (the transputer directly connected to the host) may not be part of the target network.

One or more of the skip bootables `skip`*n*`.btl` is loaded before loading the application code. Each skip bootable makes one transputer pass on everything it receives, so it becomes transparent to the software. Any code loaded is passed to the next transputer, and host communications are passed between that transputer and the host. Skipped transputers must not appear in the configuration.

The skip bootables are described in more detail in section 8.6.

4.5 Program development and support

Several tools are provided to assist in program development:

- `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` automatic build utilities.
- `imap` – the map tool which generates a memory map of the functions and variables used by the program.

4.5.1 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 the library modules that are required. Code compiled by compatible TCOFF toolsets can be mixed in the same library. Libraries and the librarian are discussed in section 6.9.

4.5.2 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. The lister can display symbolic names, code listing, the modular structure and indexing of libraries and external reference data.

For details of `ilist` see chapter 11 in the *Toolset Reference Manual.*

4.5.3 Makefile generator – `imakef`

The Makefile generator `imakef` creates Makefiles for specific program compilations. Coupled with a suitable `make` build utility, it can automate building of executable code and greatly assist with code management and version control. A `make` utility is not supplied with the toolset.

`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 error modes. These extensions are described in chapter 12 in the *Toolset Reference Manual.*

4.5.4 Memory map tool – `imap`

The memory map tool `imap` takes 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 can be output on the display screen or redirected to a file.

4.6 EPROM programming

The previous sections have described the development process for boot-from-link programs. The toolset also supports the development of boot-from-ROM applications for use in stand-alone and embedded systems.

Programs to be placed in ROM are normally 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, and then using the EPROM tool to format them for ROM:

- The configurer has command line options to specify the application is to boot from ROM. The options enable the user to specify whether all the processes are to run in RAM or whether the root processor's code is to run in ROM.
- The default behavior of the toolset is to produce a boot-from-link executable. If command line options on the configurer are used to specify that the executable is to boot from ROM, then the collector will generate an executable file which is suitable for input to the EPROM tool `ieprom`. The code will be executable from ROM or RAM as appropriate.
- The EPROM program formatter `ieprom` is provided to assist with the installation of programs into ROM. This tool generates files in a suitable format for input to ROM programmers or may be used to output ASCII hexadecimal or binary for input to the users' own ROM loaders.
- Some transputers have programmable memory interfaces which may be configured for a particular memory design.

  For IMS T400, T414, T425, T800 and T805 transputers, 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` if required. If the **MemConfig** pin is correctly wired then the transputer will automatically read this data when it is reset and use it to configure its memory interface.

  For ST20450 processors, a memory interface configuration is always required. A memory configuration can be created using `imem450` and is incorporated into the ROM by `ieprom`, as described in chapter 14.

Preparing programs for loading into EPROM is described in more detail in chapter 13. Further information about the tools, including `ieprom` can be found in the *Toolset Reference Manual*.

4.7 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. Mixed language programming is described in chapter 11.

Calling modules written in other languages is possible as long as the different calling conventions and parameter data types are respected. In all mixed language calls, parameters and return values passed must be of the correct type. Pragmas exist in the C and occam compilers to ease mixed language programming. occam library routines exist which can be used to set up static and heap areas for C code called from occam.

For further details of mixed language programming, see chapter 11.

4.8 File types and filename extensions

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

The adoption of a standard system allows filename 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 filename 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 from the filename extensions; see section 4.8.1.

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 used consistently, 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 4.2. A full list of all filename extensions used by the toolset with descriptions of the file types is given in the appendices to the accompanying *Toolset Reference Manual.*

Figure 4.3 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 4.3 highlights the differences between the different language toolsets and shows how software can be prepared for loading onto transputer hardware directly by transputer links or held in ROM.

Extension	Description
.asc	ASCII format file output by imem, showing memory configuration timings.
.bin	Binary format files produced by ieprom for loading into ROM.
.btl	Bootable code file. Created by icollect.
.btr	Executable code. 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.
.clu	Configuration linked unit. Created by occonf, the occam configurer.
.d*xx*	Map file output by the linker. The characters '*xx*' are determined by the 2nd and 3rd characters of the extension of the linker output file. For example if the linker output file takes the default extension .lku, the map file is given the extension .dku.
.epr	EPROM control file. Read by ieprom.
.h	Header files for use in C source code.
.hex	A hex dump of a file for loading onto a ROM by a custom ROM loader tool.
.ihx	Intel hex format files produced by ieprom for loading into ROM.
.inc	Include files named in #INCLUDE compiler directives for occam, or #include statements in configuration descriptions.
.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.
.mak	Makefile.
.map	Map file output by the collector.
.mem	Memory configuration file. Created and read by iemit or imem450.
.mot	Motorola 'srecord' files produced by ieprom for loading into ROM.
.m*xx*	Map file output by the compiler. The characters '*xx*' are determined by the 2nd and 3rd characters of the extension given to the compiler object file. E.g. if the compiler object file takes the default extension .tco, the map file is given the extension .mco.
.occ	occam source files. Assumed by oc, the occam compiler.
.pgm	Configuration description (source) file, read by the occam configurer occonf.
.ps	Postscript format file output by iemit or imem450 showing memory configuration timings.
.rsc	Dynamically loadable code file. Created by icollect.
.s	Assembler source files which can be read by the C assembler. (The assembler is invoked by an option to the C compiler icc).
.tco	Compiled code file. Created by all TCOFF compilers.

Table 4.2 Toolset main filename extensions

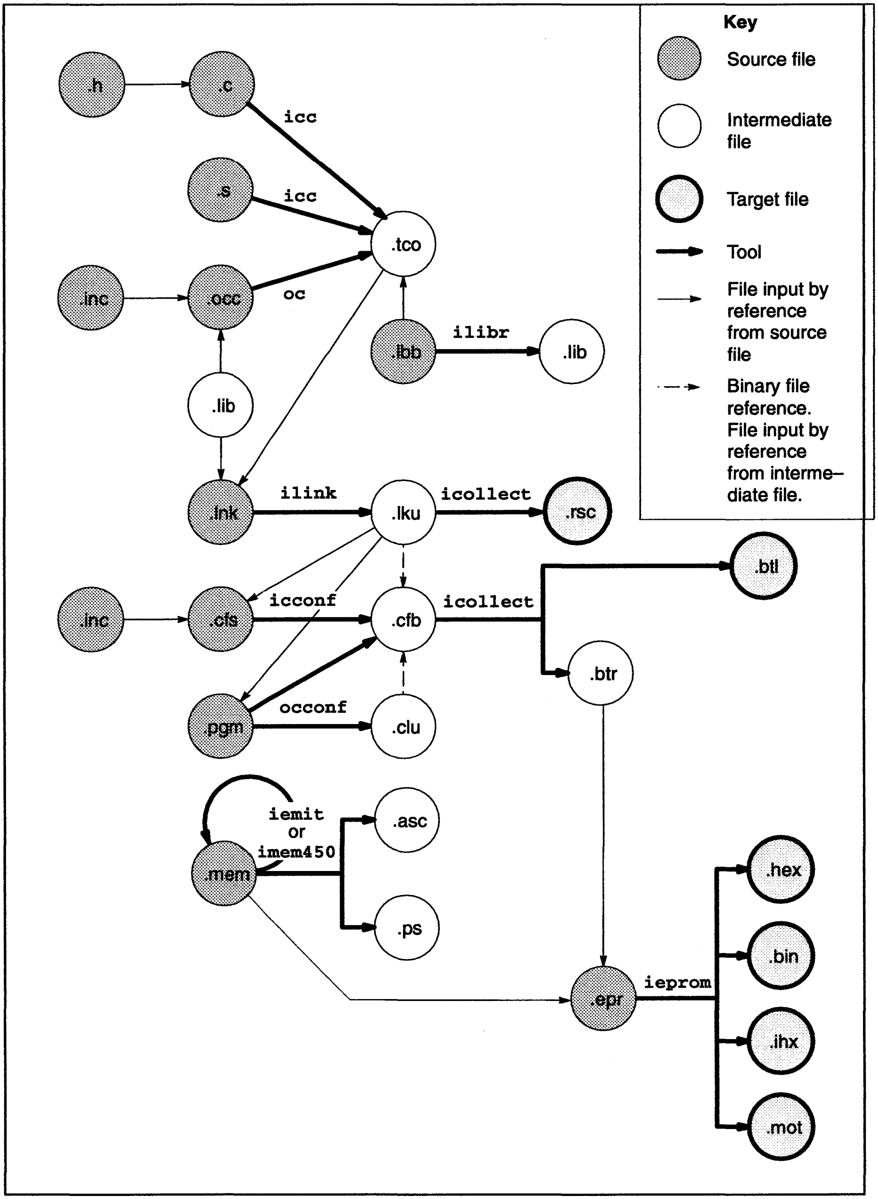


Figure 4.3 File dependencies and default filename extensions

4.8.1 Filename extensions required by `imakef`

The Makefile generator `imakef` requires a special set of filename 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 filename extensions that you must use with the `imakef` tool see chapter 12 of the accompanying *Toolset Reference Manual.*

4.9 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. 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.*

4.10 Host dependencies

The toolset uses a host to develop code which is then loaded onto a transputer or transputer network. All the tools in this toolset are executed on the host.

The toolset can be hosted on one of several different platforms, and the tools are designed to blend 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 or Solaris

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 include some minor variations in command line syntax, directory names, and environment settings such as search paths and global variables. Each is described briefly below. In addition, some extra tools are supplied with PC-hosted toolsets for running and debugging applications under Windows.

Command line syntax

The normal prefix character for tool command line options for is a minus (-). On MS-DOS based toolsets the prefix character forward slash (/) is also allowed for backwards compatibility.

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 Appendix A of the *Toolset Reference Manual.*

4.10.1 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 accept filenames which begin with a hash '#' or with two dashes '--'. These are used to identify commands and comments within linker indirect files.

Filenames prefixed by the character '@' indicate an indirect argument file; see section A.1.2 of the accompanying *Toolset Reference Manual.* Such files enable host system restrictions on command line length to be avoided.

4.10.2 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 Appendix A of the accompanying *Toolset Reference Manual.*

4.10.3 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.* The environment variables are listed in table 4.3.

Variable	Meaning
`ASERVDB`	The pathname of the AServer database, which contains descriptions of possible target hardware connections. Used by `irun`, `inquest` and profiling tools.
`IBOARDSIZE`	*For non-configured programs only.* The size (in bytes) of memory on the transputer board.
`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.
`ITERM`	The file that defines terminal keyboard and screen control codes. Used by `iemit` and `imem450`.
`TRANSPUTER`	The name of the target hardware connection, as given in the AServer Database. Used by `irun`, `inquest` and profiling tools.
toolname`ARG`	Default command line arguments. Applies to certain tools only. See section 4.10.4.

Table 4.3 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; under UNIX they are set using the `setenv` command. Examples of how to set up environment variables can be found in the *Delivery Manual* that accompanies the release.

When running applications on the target hardware with Microsoft Windows running on the host PC, some environment variables are held in a Windows initialization file. This file can be edited using the Windows launch tool `ilaunch` or from a DOS window using the `iset` tool.

4.10.4 Default command line arguments

For some tools, an environment variable can be defined to specify a default set of command line arguments. The variable name must be defined in upper case and is constructed from the tool name in upper case 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 in table 4.4.

Tool	Variable
icollect	ICOLLECTARG
iemit	IEMITARG
ieprom	IEPROMARG
ilibr	ILIBRARG
ilink	ILINKARG
ilist	ILISTARG
imem450	IMEM450ARG
oc	OCARG
occonf	OCCONFARG

Table 4.4 Environment variables for default options

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

4.11 Unsupported options

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

5 An example program

This chapter describes a more complex programming example illustrating separate compilation and showing how the program can be built for a single transputer. A simpler programming example, to get you started, is provided in chapter 3. A version configured for multiple processors is described in chapter 7.

The example program is designed for boot-from-link boards. If you have a board that boots from ROM you should set it to boot from link.

The example 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 in the `sorter` subdirectory of the `examples` directory.

5.1 Overview of the program

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

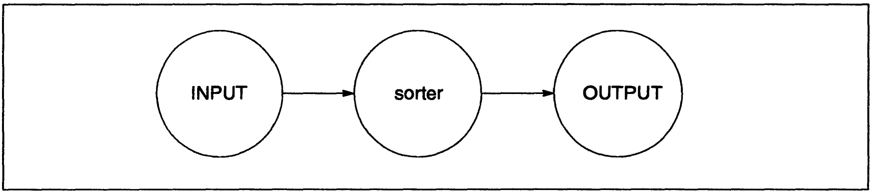


Figure 5.1 Basic structure of sorter program

Figure 5.1 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 5.2 shows the structure of the program using a pipeline.

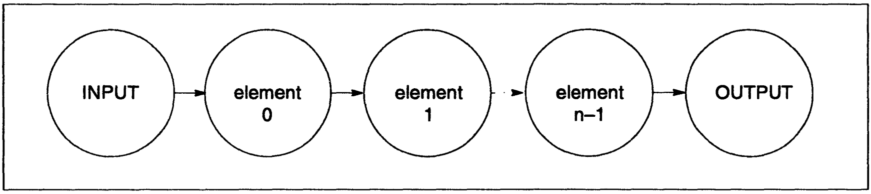


Figure 5.2 Pipeline of n elements

An obvious implementation would be to write an occam process for each process in figure 5.2, 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 procedures (PROCs) and as each of the procedures is independent we can implement each one as a separately compiled unit. The procedures 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 procedures 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 several ways round this problem:

1 use a multiplexor process for the server channels, as described in section 9.4;

2 use the AServer's multiplexing capability, as described in the *AServer Programmers' Guide*;

3 merge the two i/o processes into a single process.

Solution 3 is used here because the program accesses the server in a sequential manner, i.e. it reads a line, then displays the sorted line, then reads a line and so on. Figure 5.3 gives the final process diagram for the program.

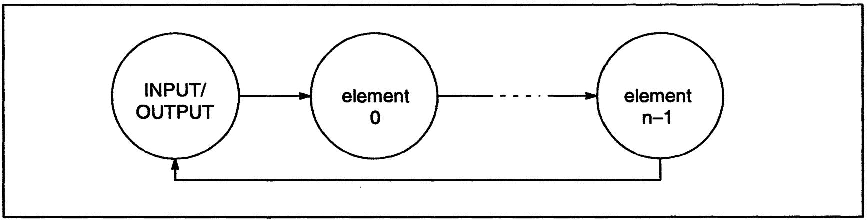


Figure 5.3 Program with combined input/output process

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

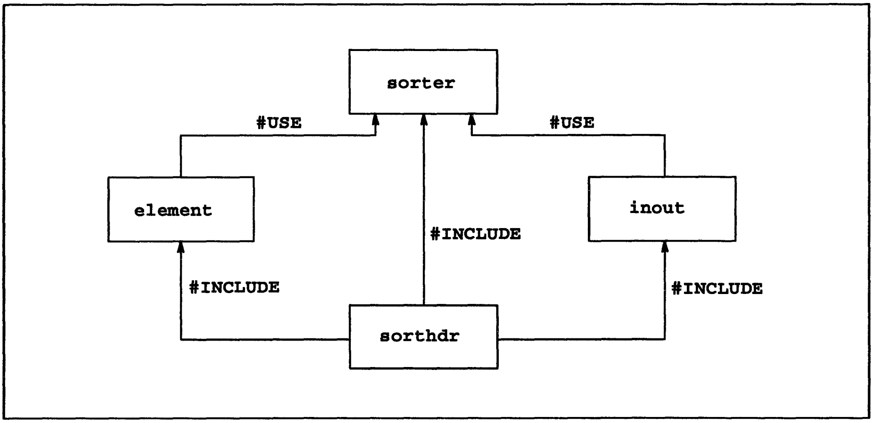


Figure 5.4 File structure of program

The source of the program is given below and is supplied in the `examples` directory. These files can be copied to a working directory.

Two system files are required to complete the program, namely the host file server library `hostio.lib` and the corresponding `.inc` file containing the host file server constants. These are automatically located using the `ISEARCH` environment variable.

5.2 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`	to mark the end of the sequence to be sorted;
`terminate`	to signal 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.

5.3 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 value character it has received 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.

5.4 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 post-mortem debugger.

The input/output process file `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
  :
```

5.5 The calling program

This process calls the input/output process in parallel with the sorter elements, in a pipeline. The calling program in file `sorter.occ` is listed below:

```
#INCLUDE "hostio.inc" --
PROC sorter (CHAN OF SP fs, ts)
  #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])
:
```

5.6 Compiling the program

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

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

```
oc element -t425
```

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 an IMS T425 in HALT mode.

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

```
oc inout -t425
```

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

Finally compile the main body using the command line:

```
oc sorter -t425
```

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

5.7 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 host i/o library used by this program. Remember the include file, `occama.lnk`, which identifies the compiler libraries, required in the linking process (see section 4.3.1).

To link the files use the following command:

```
ilink sorter.tco inout.tco element.tco hostio.lib -f occama.lnk -t425
```

The linker will create the file `sorter.lku` linked for an IMS 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.

5.8 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 `irun`. 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 1 Mbyte 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
    sorter(fs, ts)
:
```

5.9 Running the program

The bootable file can be run using `irun`. To load the program onto a transputer board use the following command:

```
irun sorter.btl
```

For more details about loading see chapter 8, and for details of `irun` see chapter 15 in the *Toolset Reference Manual.*

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

Figure 5.5 shows an example of the screen display obtained by running `sorter.btl` on a UNIX based toolset. The user enters the string `Sorter program` and terminates the program by pressing RETURN.

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

Figure 5.5 Example output produced by running `sorter.btl`

5.10 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 rebuilt.

`make` is a common utility 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` utility 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 12 in the *Toolset Reference Manual.*

6 Programming in occam

This chapter discusses some programming issues related to the facilities provided by the occam 2.1 Toolset outside the occam language. For information about programming multi-transputer networks see chapter 7.

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 2.1 Reference Manual* contains a formal definition of the language.

6.1 Host channels

Terminal and file input and output is supported by the host file server, which is resident on the host computer. A process may communicate with the host file server using the library host i/o routines. These routines require a pair of channels from and to the host file server, which are normally called `fs` and `ts` respectively. The protocol on these channels is `SP`, as defined in the system include file `hostio.inc`. Therefore a process communicating with the host file server generally has the following form:

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

Figure 6.1 shows how these channels are connected.

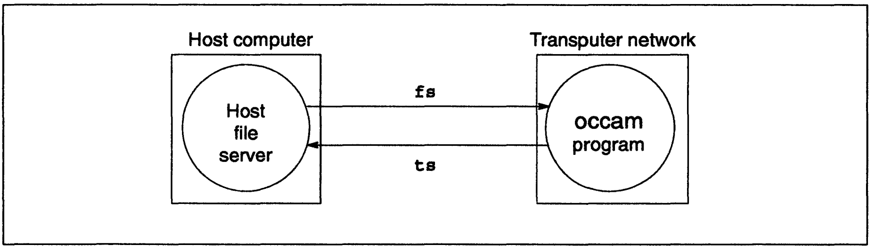


Figure 6.1 Program host input/output

Access to the host file server is via the i/o libraries, which are described in the *occam 2.1 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.

6.1.1 Interrupting programs

To interrupt an application program while it is still running, press the host system break key, usually Ctrl-C.

When the break key is pressed, the host file server terminates. This does not terminate the transputer application, although usually the application will be forced to wait for host communications. In some cases, if the host server is restarted without rebooting the transputer network, the application can resume.

6.2 occam error handling

For systems that require maximum security and reliability, the error behavior is of great concern. occam 2.1 specifies that run-time errors are to be handled in one of three ways, each suitable for different programs. This section describes the implementation of error modes in this toolset. The error mode to be used is supplied as a parameter to both the compiler and linker.

6.2.1 Error modes

The error modes provided by the toolset are listed in table 6.1.

Error option	Error mode	Description
H	HALT	Transputer halts when an error is detected.
S	STOP	Process STOPs when an error is detected.
X	UNIVERSAL	As HALT or STOP depending in **HaltOnError** flag.

Table 6.1 Compiler and linker options for selecting error mode

The first mode, called *Halt system mode* or *HALT* mode, causes any run-time error 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 users should use this mode when developing systems.

On the IMS T414, T212, T222, and M212, HALT mode does not halt the processor 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, is not described in the *occam 2.1 Reference Manual*, but is provided in the toolset to simplify occam libraries. It may behave as either HALT or STOP mode depending on the transputer's **HaltOnError** 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 incompatible and may not be combined in the same linked unit, 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 a linker 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 6.2 summarizes error mode compatibility.

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

Table 6.2 Compatibility between error modes

occam error mode UNDEFINED

The *occam 2.1 Reference Manual* describes a further error mode called *UNDEFINED*. This is not provided as an error mode in this toolset, but the same effect can be reproduced by disabling the error checking code, as described in section 6.2.2. The error checking code can be disabled in any of the error modes described above.

6.2.2 Error detection compiler options

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 much 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. The `U` option 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`.

In any situation where the compiler can detect an error without specifically inserting code, it *may* still do so. Thus an arithmetic overflows 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.

The `U` option can be used to optimize run time performance in code which is fully debugged and known to be error-free. This is equivalent to implementing UNDEFINED error mode.

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 2.1 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.

6.3 Library i/o

The occam compiler and linker use library i/o by default. The compiler will generate calls to library routines to perform channel input and output rather than using the transputer's instructions; this is referred to as using *library i/o*. Using library i/o causes a performance penalty to be incurred when virtual routing is not required. Using the command line option Y disables library i/o and results in faster code execution but the code cannot be used in a system which requires virtual routing.

Code which has library i/o disabled may call code which has library i/o enabled but not vice versa. If library i/o is disabled for any module in a program then the configurer option NV must be used to disable virtual routing throughout the program. This will prevent the program from being interactively debugged as interactive debugging requires virtual routing.

6.4 Alias and usage checking

The compiler implements the alias and usage checking rules described in the *occam 2.1 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 these rules, see Appendix C in the *occam 2.1 Toolset Language and Libraries Reference Manual.*

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 there being no aliasing in the program. If alias checking is switched off 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 2.1 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 run-time. 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 2.1 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.

6.5 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 more than 8 bytes.

This decreases the amount of stack required, which has two benefits: firstly, the offsets of variables are smaller, so 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 6.2.

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

The default allocation of 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 :
```

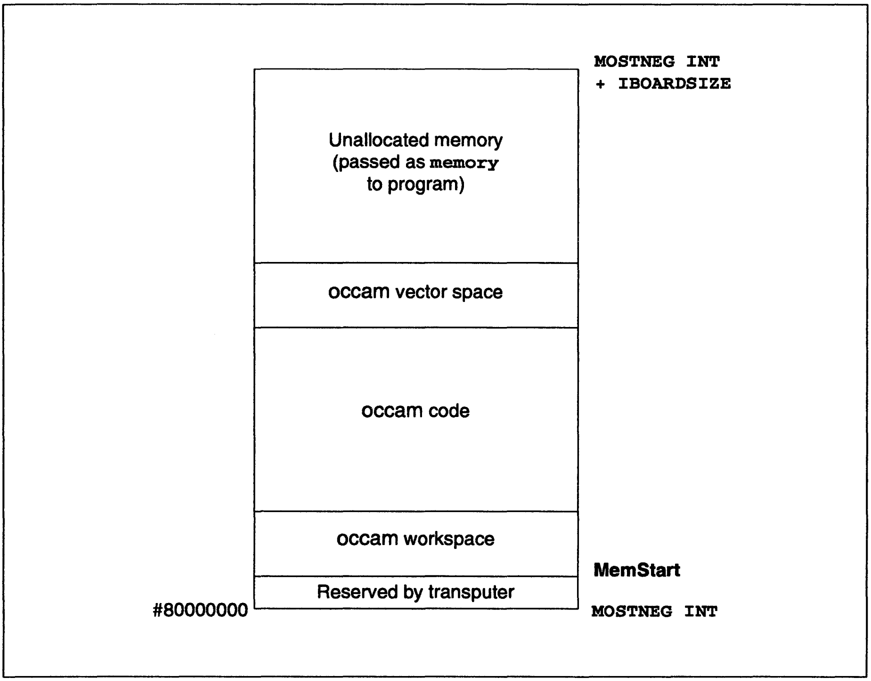


Figure 6.2 Memory allocation on a 32-bit transputer

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.

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.

6.6 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 written in quotes ("). All of the line following the closing quote may be used as for comments. All directives occupy a single line.

6.7 Separate compilation

Separate compilation reflects the hierarchical structure of occam, and the occam compiler compiles occam procedures or functions (PROCs or FUNCTIONs) or both. 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 or functions or both that are compiled together are known as a compilation unit. Each procedure 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 6.9. A separate compilation unit must be compiled before the source which references it can be compiled.

6.7.1 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 or protocol or both 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 MYPROT defined in a file myprot.inc. This might be used as follows:

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

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

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

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

  SEQ
    ... body of procedure
:
```

Since the protocol name MYPROT 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 MYPROT.

6.7.2 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 10.4 in the *Toolset Reference Manual* describes the operation of linker directives.

6.8 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`** utility 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.

The Makefile generator is used by telling it the name of the file to *build*. The tool can produce a Makefile for any type of file that can be built with the toolset tools. In order that **`imakef`** can identify file types, a different system of file extensions must be used from that used in this chapter. The filename rules for **`imakef`** are described in section 12.3 of the *Toolset Reference Manual*.

6.9 Libraries

A library is a collection of compiled procedures or functions or both. 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.

6.9.1 Selective loading

A library consists of modules where each module derived from a separately compiled unit. Each module 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 required, or compatible, transputer type or class, error mode and method of channel input/output, are read. For details of processor types and classes, see Appendix B in the *Toolset Reference Manual*, and sections 6.2 and 6.3 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 *Toolset Reference Manual*, and for information on mixing error modes see section 6.2.

6.9.2 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 9 of the *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
```

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 12 of the *Toolset Reference Manual.*

7 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.

7.1 Introduction to configuration

This section introduces the concept of configuration and describes the configuration model. It also provides some background information about network structure and the communication facilities supported. Example configurations are given at the end of this chapter as well as in the *Getting started* chapter of this manual.

The INQUEST network analyzer `rspy` can be used to generate a hardware description in a form suitable for use in configuration files. For further details of `rspy` see the *INQUEST User and Reference Manual.*

7.1.1 What is configuration?

An application generally contains a number of communicating processes which may be designed to run in parallel. In a single transputer system, all the application software will be loaded onto one transputer; in a multiprocessor system the application processes may be distributed over several transputers.

Configuration is the process of defining the software and hardware components of a system, describing how the software is placed on the hardware, and defining the communications requirements of the system.

The application software is described in terms of its component processes and how they communicate. The way that processes are placed on the hardware is described as the *mapping* of software to hardware. The hardware is described in terms of the number and type of transputers, the amount of memory on each transputer and how they are connected.

Configuration is achieved by writing a configuration description in the occam configuration language. A configuration description is a text file created by the user, which is processed by the configurer tool `occonf` to generate a configuration data file. This 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 same software description to be used on a variety of alternative hardware networks. Likewise the same hardware description may be used in a variety of configurations describing different programs that may be run on the same hardware.

7.1.2 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*.

7.2 Configuration model

Each transputer has up to four hardware links. These links can be connected to one another to form communication links between transputers. The transputers and their connections are known as a *network*, and the individual devices are *nodes* of the network. Links can also be used in conjunction with link adaptors, to connect peripheral devices to the network.

The configuration model consists of the following:

- A hardware description, defining the transputers in the network and how they are connected.
- A software description, defining the processes and how they are connected via channels, in the form of an occam process.
- A mapping between the processes and channels of the software and the transputers and transputer link connections respectively of the network.

The software description takes the form of an occam process with at least as many parallel sub-processes as there are 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 physical processors declared in the hardware description, or may be logical processors mapped onto the physical processors in a separate mapping description. 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 to determine what code is to be loaded onto what processor, and to choose the mapping of channels onto links between processors.

Channels may also be used to connect to external hardware, such as the development host or peripherals connected by means of link adaptors. External objects of this kind are declared as `EDGE`s in the hardware description.

The hardware connections, to external edges or between processors, may optionally be associated with a name, declared as an ARC. This allows explicit mappings of channels onto these connections.

7.2.1 Configuration tools

The following tools are provided for combining linked units into executable code: `occonf`, `icollect`, `iemit`, `imem450` and `ieprom`. They are described in the *Toolset Reference Manual*.

`occonf` and `icollect` are used in the main development process of configuring an application to run on a transputer network and are discussed in this chapter. Depending on the transputer type, `iemit` or `imem450` may be used to generate a memory configuration file to configure the transputer's external memory interface. `ieprom` is used to load a configured application and a memory configuration file into ROM. Further information about EPROM programming and configuring the ST20450 memory interface is given in chapters 13 and 14 respectively.

7.2.2 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, as the configurer uses occam rules to evaluate these statements. Appendix G of the *Toolset Reference Manual* defines the syntax of the occam configuration language.

Configuration declarations introduce physical processors, arcs and edges of the hardware, hardware 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 7.1.

Statement	Description
NODE	Declares a processor (*node* of a graph). A processor is considered to be *physical* if it is defined as part of the hardware description, or *logical* if it is defined as part of the software description and mapped to a physical processor as part of the mapping.
ARC	Declares a name for a hardware connection (*arc* of a graph) between processors (using transputer links) or between a processor and an external object. Connections need not be associated with ARCs unless channels are required to be explicitly placed on particular links.
EDGE	Declares an object external to the transputer network. An external object may be the host, or a peripheral connected via a link adaptor e.g. a joystick, disc drive.
NETWORK	A construct containing the connections and attribute settings of previously declared NODEs (physical processors).
MAPPING	A construct containing mappings between previously declared logical processors and physical processors.
CONFIG	A construct containing the software description.

Table 7.1 Configuration description declarations

Arrays of NODEs, EDGEs, and ARCs may be declared. A configuration description includes one NETWORK, one CONFIG and, optionally, one MAPPING construct. 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 construct is optionally named by an identifier following the 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 7.2.

Statement	Description
SET	Defines values for NODE attributes.
CONNECT	Defines a connection between two EDGEs, either the links of two NODEs or between a link of a NODE and an EDGE. See section 7.3.
MAP	Defines the mapping of a logical processor NODE onto a physical processor 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 7.2 Configuration description statements

A MAP statement may only appear within the MAPPING construct. A CONNECT statement may only appear within the NETWORK construct. A SET statement may appear in either the MAPPING or the NETWORK constructs.

7.2.3 Importing code and source files

Compiled and linked 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 (hardware, software and mapping) 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.

7.2.4 Overall structure of a configuration description

A configuration description consists of two or three parts; a hardware description, a software 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.

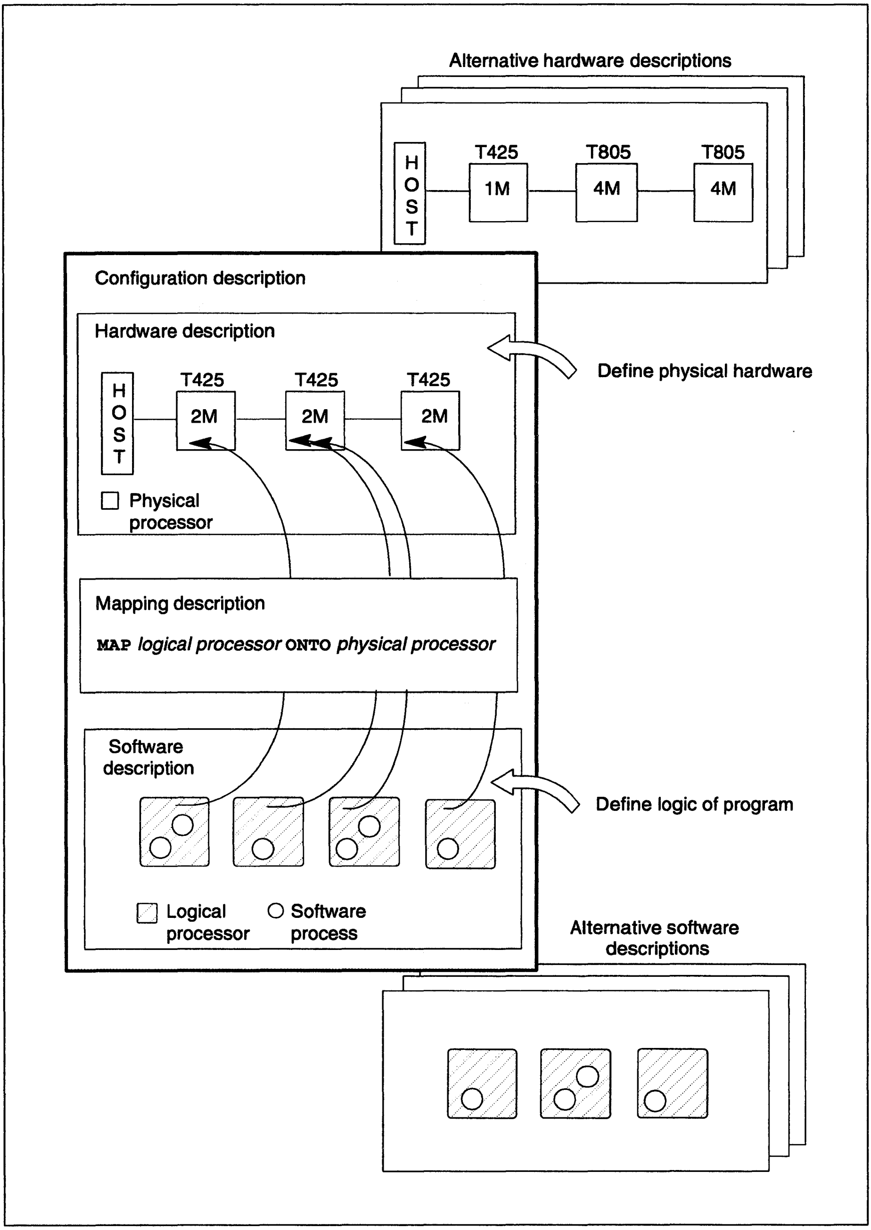


Figure 7.1 Configuration using logical processors

The software description is essentially 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 program to execute on a different hardware network. The mapping section enables this to be performed without modifying the software description in any way, by flexibly mapping the *logical* processors onto the *physical* processors, as shown in figure 7.1.

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 := "T425", 1 * M)
    SET worker.p (type, memsize := "T805", 4 * M)
    CONNECT root.p[link][3] TO worker.p[link][0]


-- mapping description
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 7.2.

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

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

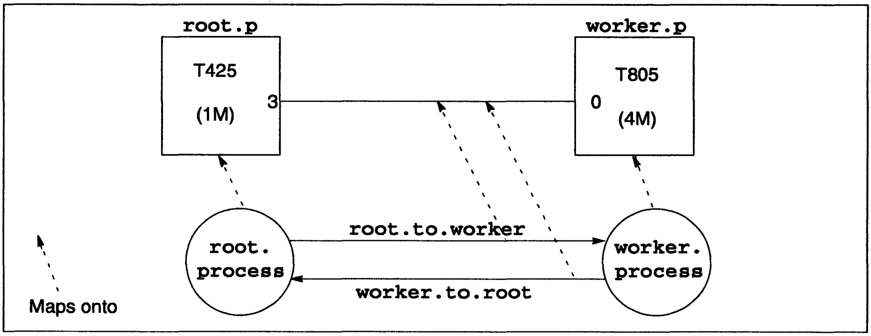


Figure 7.2 Mapping of software onto hardware

To derive this shortened description from 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 := "T425", 1 * M)
    SET worker (type, memsize := "T805", 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)
:
```

7.3 Hardware description

The configuration description for the hardware defines what processors are present and how they are connected by means of their links. A simple example was shown in chapter 3 *Getting Started*, which defines a single transputer connected to the host.

A hardware description consists of physical processors of type NODE connected by their links with the CONNECT statement. A processor has attributes which are used to define the processor type, memory size and other characteristics.

7.3.1 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
```

7.3.2 NODE attributes

A NODE representing a physical processor has a set of attributes, analogous to fields of a record. Attributes may be values (such as processor type or memory size) or edges (such as the links).

The attribute names, which are predeclared by the configurer, do not follow the occam scope rules; they are only recognized in the context of a SET statement for value attributes or a CONNECT statement for edge attributes.

Value attributes

An integer, boolean or string value can be assigned to a value attribute using a SET statement. The value can be any occam expression of the appropriate type. For example, the following statement sets the type attribute of proc to T805:

```
SET proc ( type := "T805")
```

No attribute may be set more than once for each processor.

Edge attributes

An edge attribute is referenced by subscripting the name of the node with the name of the attribute. Edge attributes are arrays of type []EDGE, so a subscript is required to reference an element of the array. For example, link 2 of proc is referenced by:

```
proc[link][2]
```

Edges are connected using a CONNECT statement. For example, the following specifies that link 0 of proc1 is connected to link 3 of proc2:

```
CONNECT proc1[link][0] TO proc2[link][3]
```

Processor attributes

The attributes of processors are listed in tables 7.3 and 7.4. The attributes listed in table 7.3 are used in the hardware description and are described below. The attributes in table 7.4 are used in the mapping description and are described in more detail in chapter 10. All the attributes are value attributes except `link` which is an edge attribute.

`type` and `memsize` must be defined for all processors. The `type` attribute must be set to a string (of any length) whose contents describe the processor type. Trailing spaces at the end of the processor type are ignored.

The `memsize` attribute must be set to the amount of usable memory (on-chip + external memory) available to the processor, expressed in bytes, which is to be used by the configurer to place the code and data of processes. `memsize` defines the size of this memory as a contiguous region of memory starting at the most negative address, `MOSTNEG INT`. Regions of memory elsewhere in the address space are not included. The constants `K` and `M`, defined in `occonf.inc`, can be used to specify Kbytes and Mbytes. The size specified must be positive and should represent a whole number of words for the processor. The top of the memory region specified by `memsize` must be greater than or equal to the value of **LoadStart**; details of **LoadStart** are given in chapter 10.

The `memstart` and `numlinks` attributes are only used with processors whose `type` is set to `ST20`. `memstart` specifies the location of **MemStart** as an absolute address, which must be word aligned. `numlinks` specifies the number of links, and hence the size of the `link` array. It is good practice to set these attributes for ST20 processors, as the location of **MemStart** and the number of links depends on the ST20 variant in use.

Attribute	Type	Description
`link`	`[]EDGE`	The links of the processor. This attribute is only defined if `type` has already been defined, and if `type` is `ST20` then `numlinks` should also be defined and is the size of the array.
`memsize`	`INT` *Mandatory*	Defines the size in bytes of the contiguous region of memory available to the processor, starting from `MOSTNEG INT`.
`memstart`	`INT`	Defines the location of **MemStart** for processors of type `ST20` only. It is expressed as an absolute address, which must be word aligned. The default value is **MemStart** for the ST20450.
`numlinks`	`INT`	The number of links available on the processor for processors of type `ST20` only. Valid values are in the range 1 to 4 inclusive. The default value is 4.
`romsize`	`INT`	Defines the size in bytes of ROM attached to the processor. Mandatory if `root` is `TRUE`.
`root`	`BOOL`	Defines the root processor if there is no host connection. Takes the values `TRUE` or `FALSE`. The default value is `FALSE`.
`type`	`[]BYTE` *Mandatory*	Defines the processor type as a string. Processor types supported are: `T212`, `T222`, `T225`, `M212`, `T400`, `T414`, `T425`, `T450`, `ST20`, `T800`, `T801` and `T805`.

Table 7.3 Hardware description attributes

If a network is configured to be booted from ROM, the attribute `root` must be set to `TRUE` for one processor only. The attribute `romsize` must also 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.

Additional processor attributes can be set in the `MAPPING` section. These support the following features:

- relative ordering of code, workspace and vector space (`order.code`, `order.ws` and `order.vs`);
- placing code, workspace, and vector space (`location.code`, `location.ws` and `location.vs`);
- reserving memory (`reserved`);
- fine-tuning of software virtual routing (`routecost`, `tolerance` and `linkquota`);
- disabling the INQUEST tools for specific processors (`nodebug` and `noprofile`).

Attribute	Type	Description
`linkquota`	`INT`	Defines the maximum number of links on a processor to be used by virtual routing.
`location.code`	`INT`	Defines the absolute address at which the code for a program on a processor should be placed in memory.
`location.vs`	`INT`	Defines the absolute address at which the vector space for a program on a processor should be placed in memory (if it exists).
`location.ws`	`INT`	Defines the absolute address at which the workspace for a program on a processor (stack) should be placed in memory.
`nodebug`	`BOOL`	For use with the INQUEST debugger. It informs the debugger that the processor is not to be debugged. It takes the values `TRUE` or `FALSE`; the default is `FALSE` if the `GA` configurer option is used, and `TRUE` if the `GD` option is used.
`noprofile`	`BOOL`	For use with the INQUEST profiling tools. It informs the profilers that the processor is not to be profiled. Takes the values `TRUE` or `FALSE`; the default is `FALSE`.
`order.code`	`INT`	Defines the ordering priority of the code for a program on a processor in memory. The default value is 0.
`order.vs`	`INT`	Defines the ordering priority of the vector space for a program on a processor in memory. The default value is 0.
`order.ws`	`INT`	Defines the ordering priority of the workspace for a program on a processor in memory. The default value is 0.
`reserved`	`INT`	Defines the size in bytes of a contiguous block of memory starting at `MOSTNEG INT` to be reserved for code and data placement.
`routecost`	`INT`	Defines the cost of using a processor in the network for virtual routing.
`tolerance`	`INT`	Defines the level of usage of a particular processor for load-sharing routing paths.

Table 7.4 Mapping description attributes

7.3.3 NETWORK description

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

To define a single processor, the SET statement provides values for the processor's attributes in the style of a multiple assignment, for example:

```
NETWORK single
  SET processor ( type, memsize := "T805", 1024*1024 )
:
```

A sequence of SET statements must be enclosed in a DO construct, for example:

```
NETWORK single
  DO
    SET processor ( type    := "T805" )
    SET processor ( memsize := 1024*1024 )
:
```

The DO construct does not imply any particular ordering, so there is no absolute constraint on the order in which attributes may be defined. However, it is considered good occam style to set the processor attributes before other statements such as CONNECT statements.

Hardware description constructs

IF, SKIP and STOP may be used in DO constructs. The syntax is the same as for occam.

Processors are connected by means of CONNECT statements. Each CONNECT statement connects a pair of edges, each of which may be an edge attribute of a node (such as the link of a processor) or a network EDGE. For example:

```
VAL K IS 1024:
NETWORK pair.from.ROM
  DO
    SET proc1 ( type, memsize := "T805", 2048 * K)
    SET proc1 ( root, romsize := TRUE, 256 * K)
    SET proc2 ( type, memsize := "T425", 1024 * K)
    CONNECT proc1[link][0] TO proc2[link][3]
:
```

The order of the two edges in a CONNECT statement is not significant.

Arrays of processors do not need to be all set to the same type or the same value for other attributes. The attributes can be set by using DO replicators within the NETWORK construct, and by using conditionals, as in this example:

```
NETWORK pipe
  DO
    DO i = 0 FOR 100
      IF
        (i \ 4) = 0
          SET processor[i] (type, memsize := "T805",
                                    4 * (1024 * 1024) )
        TRUE
          SET processor[i] (type, memsize := "T425",
                                    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 processor.type IS ["T425", "T425", "T425", "T805"] :
NETWORK fancy   -- every fourth processor is different!
  DO i = 0 FOR SIZE processor
    SET processor[i] ( type := processor.type[i \ 4] )
:
```

7.3.4 Declaring EDGEs

EDGEs define the external interfaces of a NETWORK, often implemented by link adaptors, such as an interface to a host or peripheral. 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]
:
```

7.3.5 Declaring ARCs

It is necessary to name any connection to an EDGE so that channels can be placed on it. It may also be necessary under certain circumstances to name a connection between two processors to allow the explicit placement of channels onto hardware connections. This is not normally necessary, because the configurer can place channels between processors onto links automatically. Explicit placement may be needed, for example, if there are multiple links between two processors, and one link runs at a different data rate from the others.

Named connections are called ARCs, and are declared as though they were occam data types. They are associated with a connection by adding a WITH clause to the end of a CONNECT statement. For example:

```
EDGE joystick :
ARC  link.to.joystick :
NODE controller :
NETWORK n
  DO
    SET controller (type, memsize := "T225", 64 * 1024)
    CONNECT controller[link][2] TO joystick WITH
                                          link.to.joystick
:
```

7.3.6 Abbreviations

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

```
[10]NODE pipe :
NETWORK pipeline
  DO i = 0 FOR 10
    NODE this IS pipe[i] :
    SET  this (type, memsize := "T425", 1024*1024)
:
```

A link of a processor may be abbreviated as an EDGE, for example:

```
[10]NODE pipe :
NETWORK pipeline
  DO
    DO i = 0 FOR 10
      SET pipe[i] (type, memsize := "T425", 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 :
```

7.3.7 Host connection

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

```
NODE single :
ARC  hostlink :
NETWORK B008
  DO
    SET single (type, memsize := "T805", 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.

7.3.8 Example – a single processor connected to the host

```
#INCLUDE "occonf.inc"
NODE single:
ARC hostlink:
NETWORK B008
  DO
    SET single (type, memsize := "T425", 2 * M)
    CONNECT single[link][0] TO HOST WITH hostlink
:
```

This configuration is illustrated in figure 7.3.

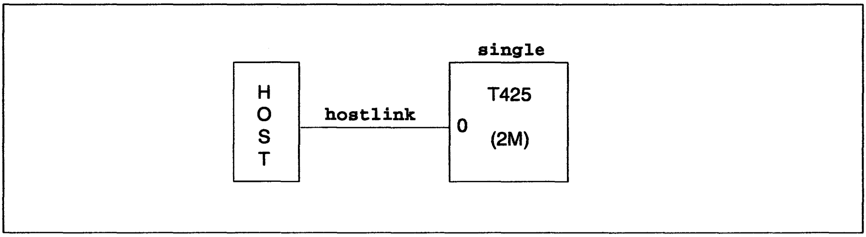


Figure 7.3 Example of host connection

7.3.9 Example – a simple pipeline

This example shows a simple pipeline, where the root processor has a different memory size from the other processors.

```
#INCLUDE "occonf.inc"
[p]NODE pipe:
ARC hostlink:
NETWORK simple.pipe
  DO
    SET pipe[0] (type, memsize := "T805", 2*M)
    DO i = 1 FOR p-1
      SET pipe[i] (type, memsize := "T805", 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 7.4.

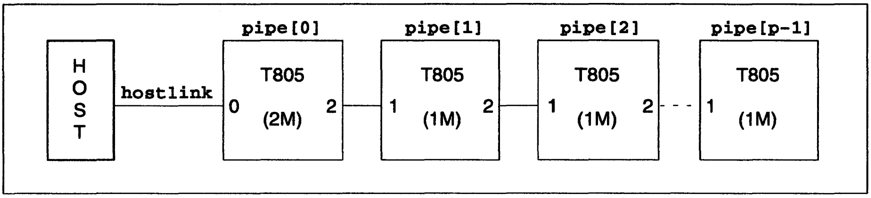


Figure 7.4 Simple pipeline with different processor memory sizes

7.3.10 Example – a 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 := "T425", 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 := "T805", 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]
  :
```

7.4 Software description

The software description is introduced by the CONFIG construct 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.

The *logical* processor names must be introduced first by means of NODE declarations. These look identical to those used in the hardware description, but have no attributes. Since these must be visible to a following MAPPING section, they must be declared *outside* the CONFIG construct. Channels which are to be mapped onto ARCs by mapping statements within a MAPPING section must also be declared outside the CONFIG construct.

A PROCESSOR statement associates the process instance 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 associated with a 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.

7.4.1 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. If the two linked units are referenced directly within the software description by `#USE` directives with nested scope, then one of them will 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 version of the process for each `PROCESSOR` statement it finds.

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

7.4.2 Example

The following example of a software description, is for the pipeline sorter program introduced in chapter 5. The example is developed to show the complete configuration description for the program, in section 7.6. Figure 7.5 illustrates the mapping of the software processes onto a network of logical processors.

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

This example names a single process `inout` associated with the logical processor `inout.l` and an array of processes `sort.element` associated with the array of logical processors `pipe.element.l`. The program may be mapped onto any hardware configuration which matches the logical network as described by the software description and which includes an `ARC` declaration for the host connection `hostlink`.

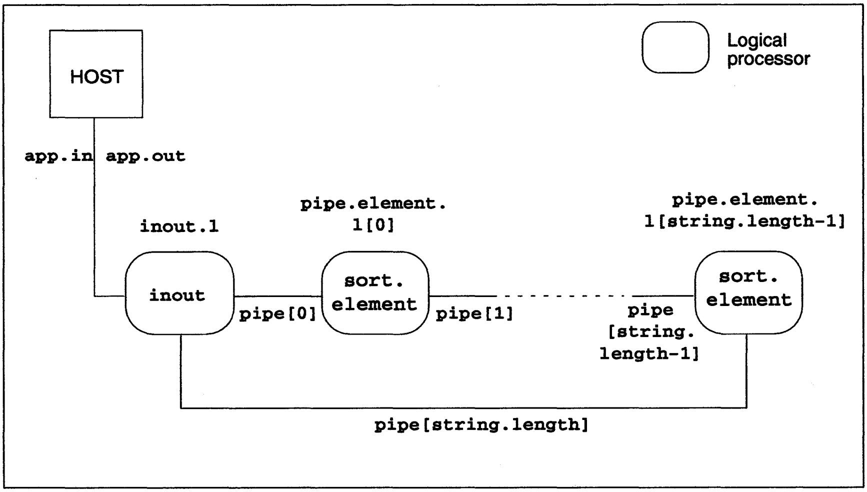


Figure 7.5 Pipeline sorter -- mapping processes onto processors

7.5 Mapping description

A MAPPING construct is required if the user has declared logical processors. The MAPPING construct enables the user to map 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 a physical processor. The MAPPING construct also enables the user to map channels onto links.

The priority at which a process runs may be determined as part of the mapping, if that 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 scope rules, logical processors must be declared (as NODEs) before the MAPPING construct. Each logical processor 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. Similar rules apply for channels and ARCs.

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

7.5.1 Mapping processors

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
:
```

The optional PRI clause to the MAP statement is available if the priority of a process or processes associated with a logical processor are required to be determined by the mapping. The argument to PRI can be either 0 to indicate *high* priority, or 1 to indicate *low* priority. For example:

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

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

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

7.5.2 Mapping channels

At the configuration level, channels are defined in the software description by occam CHAN declarations. Channels connecting processes do not need to be mapped, as the configurer can derive how to implement a channel from the mapping of the processes that use the channel.

7.5 Mapping description

The user may wish to place a channel explicitly to override the default configurer placement. In general, channels should not be explicitly placed on links, as this enables the configurer to implement channels using routing and multiplexing software where applicable.

Explicit channel mapping

A channel connecting a processor to an external EDGE must be explicitly mapped onto an ARC which connects to that EDGE. A channel may also be explicitly mapped if the user wants to override the default allocation.

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. The ARC must connect EDGEs of the processors onto which are mapped the processes which use the channel.

Channels may be mapped onto 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.

Direct and virtual channels

A channel may be implemented in one of three ways by the configurer:

- A *soft channel* is a channel which communicates between processes running on the same processor.
- A *direct channel* is one of up to two channels (one in each direction) placed on a single link between adjacent processors or between a processor and an edge.

- A *virtual channel* is a channel between two processes on different processors which is not a direct channel, i.e. it may be routed through intermediate processors, or may be multiplexed with other virtual channels. A virtual channel is placed on a *virtual link*.

No action is required at configuration time to place the soft channels within an application; they are local to a processor and so the configurer need take no action.

Other channels are mapped automatically onto the available hardware links. If channels are between processes on adjacent processors, then they can be placed directly onto the link or links that join the processors. Direct channels occur when only one or two channels (one in each direction) are using a link. A channel connecting an edge to a process is always a direct channel. Direct channels may be automatically allocated by the configurer or the user may specifically place up to two channels on a link. Channels placed in the mapping description are always direct channels.

If more than one pair of channels need to use a link, then all channels needing to use the link will be virtual. Any channel between processes on processors that are not directly connected will also be virtual. The configurer can implement many virtual channels over a single link as well as channels between processes on non-adjacent processors. They are implemented by software virtual routing processes added automatically, as required, by the configurer for multiplexing many channels over a link and for through-routing channels between non-adjacent processors.

Virtual channels enable an application program to run on most network topologies irrespective of the number of links connecting the processors. The configurer can form virtual channels that span up to 24 hops across the target network, i.e. up to 24 intermediate processors may be used for routing a 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 require multiplexing and demultiplexing or routing processes or both, which incur a run-time overhead in memory and CPU resources. The bootable file generated will be smaller if virtual routing processes are not included. In certain performance critical applications it may be important to avoid the overhead incurred when using virtual channels by forcing channels to be direct or by using the attributes described in chapter 10 to minimize the overhead on the most critical channels.

In each direction, a link may carry either one explicitly placed channel or any number of virtual channels. A pair of virtual channels (one in each direction) may be routed by the configurer via different links.

Note: The INQUEST interactive debugger uses hidden debugging channels to pass debugger commands and data between the application and the host. In order to implement these channels, some channels that would otherwise be direct are implemented as virtual. In this case, it should be assumed that *all* channels that connect processes on different processors may be virtual.

Direct channel communication

If all channels in a linked unit are intended to be direct channels, then the code must be compiled and linked with the Y option. Code built in this way cannot be configured unless the NV option is specified to occonf.

The NV option should also be used if it is required that the configuration does not use any virtual routing, e.g. for performance reasons. This disables the configurer from using virtual routing processes. The configurer will generate an error message if configuration is not possible, in which case the configuration should be modified to ensure that all channels can be placed.

If the NV option is specified then the program cannot be interactively debugged with the INQUEST debugger.

7.5.3 Mapping without a MAPPING section

Channels can also be mapped onto ARCs outside the MAPPING construct, using the PLACE statement. This is known as *channel allocation*. Any channel in scope at the point where a process is associated with a processor is available for explicit placement on an ARC declared in the hardware description.

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.

7.5.4 Mapping example – pipeline sorter on a single processor

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

7.5.5 Mapping example – pipeline sorter on a ring of processors

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

7.6 Example – a pipeline sorter on four transputers

This section describes how the pipeline sorter program first described in chapter 5 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 instructions about how to compile, configure and run the program.

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

`sorthdr.inc`	the shared protocols.
`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 chapter 5.

`sortconf.pgm` describes the hardware and software networks and maps the software to the hardware. The software description is imported from 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 7.6. 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 7.6.

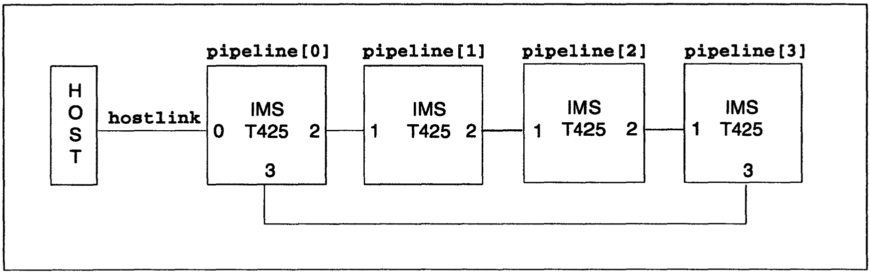


Figure 7.6 Pipeline of four transputers

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.

```
--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:
ARC Hostlink:


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

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

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

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


--mapping description
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[0] PRI HIGH

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

    MAP pipe.element.p[elements.on.root-1] ONTO pipeline[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[j+1] PRI HIGH
        DO i = first.element.here + 1 FOR elements.per.transputer - 2
        MAP pipe.element.p[i] ONTO pipeline[j+1] PRI LOW
        MAP pipe.element.p[last.element.here] ONTO
                                   pipeline[j+1] PRI HIGH
:
```

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

In the mapping description 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 7.7. 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.

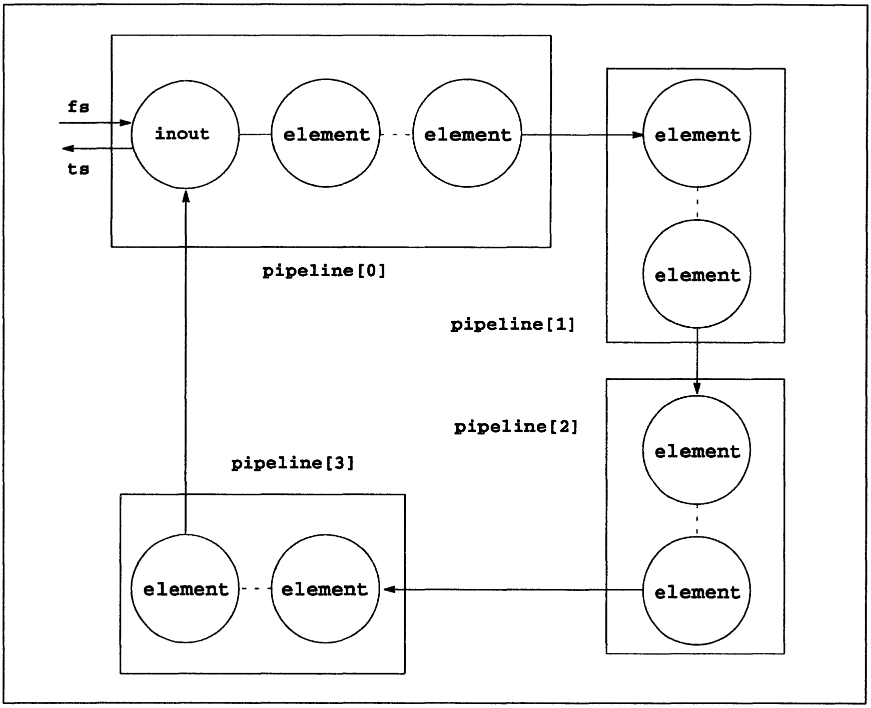


Figure 7.7 Pipeline sorter processes

7.6.1 Building the program

Building the single transputer versions of this program is described in chapter 5. This section describes how the steps to build a multi-processor version differ from the single processor version.

Compiling

The components of the program must be compiled in the same way as described in chapter 5.

Linking

The compiled files must be individually linked, since each represents a process which is to be placed on a different processor. Each must be linked with any files they reference and the compiler libraries. Each linking operation creates a unit of code which may be loaded onto the processor, according to the configuration defined in the configuration description.

Configuration

Configure the file `sortconf.pgm` which defines both the communication channels between the processes and how they should be mapped onto the hardware:

```
   occonf sortconf.pgm
```

This creates an output file called `sortconf.cfb`.

Collecting

After configuration, the program must be made executable by collecting the file `sortconf.cfb` using `icollect`:

```
   icollect sortconf.cfb
```

This creates the bootable file `sortconf.btl`.

7.6.2 Running the program

Load and serve the bootable file on the transputer network using the application loader `irun`:

```
   irun sortconf.btl
```

The program will then sort each line of input until terminated by a blank line.

7.6.3 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` utility. 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 `.c`*xx*.

`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 *Toolset Reference Manual.*

For more information about **imakef** see chapter 12 in the *Toolset Reference Manual.*

7.6.4 Other configuration examples

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

7.7 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:
   - Describing your hardware network.
   - Inserting **PROCESSOR** statements into your program and adding any necessary mapping description.
3. Compile all the separate compilation procedures that form the code for each transputer in a bottom-up fashion.
4. Link each configuration procedure with its component parts into a file with the name used in **#USE** directives in the configuration source file.
5. Run the configurer on the configuration description file.
6. Collect the code using **icollect**.
7. Load the program into the network using the host file server.

Steps 3 to 6 can be automated by using **imakef** and a suitable **make** utility.

8 Loading application programs

This chapter explains how to load and run application programs on the target hardware from a host machine. It briefly describes the format of loadable programs and introduces the `irun`, which is used for loading programs, the AServer, the AServer database and the skip bootables. The chapter ends with the use of the network analyzer `rspy` to clear error flags.

8.1 Introduction

Transputer programs may be loaded from a host onto transputer boards by the `irun` application loader tool. The code and data are sent from the host to the target network using transputer links, so this process is called *booting from link*, to distinguish from booting from a ROM.

The executable file consists of code to which bootstrap information has been added to make the program self-booting on the target network. This self-booting executable code is known as *bootable* code and the file in which it is held by the host is known as a *bootable file*.

A bootable file is generated by `icollect` from a configuration data file, using the linked units referred to by the configuration. The default filename extension for bootable host files is `.btl`. The bootable file is constructed such that copying it to the link will boot the network automatically. Code is installed on each processor using processor and distribution information embedded in the bootable file.

Transputer programs may also be loaded from ROM. The `ieprom` tool is used to generate the file to be loaded into the ROM from a bootable-from-ROM file, which usually has a `.btr` extension. See chapter 13 for a description of building ROM files. This chapter is concerned only with booting from link.

8.2 Tools for loading

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

- `irun` – the host file server and application loader.

  `irun` loads the bootable file onto the single transputer or transputer network and provides communication with the host. `irun` can be started from a command line, for example:

  ```
  irun myprog.btl
  ```

- `skip`*n*`.btl` – the skip loading bootables.

  The skip bootables allow a program to be loaded over one or more transputers onto a target network. One or more skip bootables are loaded and run immediately prior to loading the application. The skip bootables start up special route-through processes on the transputers that transfer data between the rest of the network and the host system. Skip bootables are loaded by `irun`.

8.3 The boot from link loading mechanism

The `irun` 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 bootstrap code for the transputers in the network is sent first. The code is propagated through the network as individual processors load neighboring processors. The bootstrap code on each processor runs and performs the following tasks:

1 initialize the processor,

2 pass code and data from the host to adjoining processors,

3 read in application program code and start to execute it.

Thus, after all the transputers in the network have been booted, application program code is sent to the individual processors. The allocation of processes to processors is determined by the configuration file.

When the code is loaded into the transputers' memories, the program starts running and can communicate with the host using the standard library routines for input and output.

The program continues to run until either:

- an error occurs,
- the program terminates naturally.

The server can be terminated, for example by pressing the interrupt key (usually CTRL-C or CTRL-BREAK). Terminating the server will not stop the program running on the transputer, but any processes on the transputer which attempt to communicate with the host will wait indefinitely for the host server to respond. This may eventually cause the whole program to halt as other processes become dependent on this communication. The program may be able to continue if the server is restarted.

If a skip bootable is used, the first transputer in the network becomes transparent to the software. Code apparently loaded onto the first processor will in fact be sent to the next processor. That processor may then communicate with the host as if it were directly connected.

8.3.1 Initializing the ST20450 memory interface

The memory interface of the ST20450 must be initialized before the external memory can be accessed. This may be done by a ROM or by loading a memory bootable file before loading the application bootable. Building and installing a memory bootable file is described in chapter 14. A T450 TRAM has a ROM containing code which initializes the memory interface and then simulates booting from link.

The memory interfaces of earlier IMS T2xx/T4xx/T8xx transputers can only be configured by a ROM or by wiring the transputer.

8.4 Boards and subnetworks

A transputer board may be designed to boot from link or to boot from ROM. Each transputer can be wired either to boot initially from ROM or to wait for code to arrive on a link. In a multi-transputer system, it is usual either for all transputers to boot from link (a Boot-from-link system) or for one transputer to boot from ROM which then boots the other transputers by link.

Boot-from-link systems are used as development systems and for systems that work under the control of a host. `irun` loads the application code down the hardware serial link that connects the root transputer to the host, known as the host link. Programs intended to run on boot-from-link boards must consist of bootable code, such as that generated by `icollect`. An example of a boot-from-link board supplied by SGS-THOMSON is the IMS B008 PC TRAM motherboard (with appropriate TRAMs).

Boot-from-ROM systems are used for stand-alone applications such as embedded systems. One transputer, known as the root transputer, on power-up starts executing code from the fixed address #7FFFFFFE (32-bit transputers) or #7FFE (16-bit transputers), which is the top byte but one of the address space. Included in the ROM is code for any other transputers and a loader to distribute that code.

Some other combinations are supported. For example, general purpose ST20450 modules may boot from ROM to initialize the memory interface from ROM code. The ROM code should leave the ST20450 waiting for a bootable application to arrive on a link. The module then behaves as if it were a boot-from-link module.

8.4.1 System services wiring

System services or *subsystem* wiring is the way in which board level control signals are connected, in order to simplify the control of multi-transputer systems built from standard modules.

The control signals are used to reset the transputers, detect when an error has occurred and prepare a system for post-mortem debugging. These three functions are performed by three signals, namely **Reset**, **Analyse**, and **Error**. Together these are known as the *system services*. All SGS-THOMSON transputer boards use a common scheme for propagating these signals to other subnetworks, as described in the following paragraphs, so that a single connection can control or read one of these three signals for an entire system or subsystem.

The system service signals on each board are connected to ports for connection to other boards or a host. The ports are of three types, namely *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 send 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, so the whole network can be controlled by a single signal.

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

Figure 8.1 shows a typical multi-transputer system wired so that the whole system is controlled by the host. This is the normal wiring for development and hosted systems. Each of the three transputer boards has two transputers, both connected to the subsystem wiring on the board, which is also connected to an Up port and a Down port. The host has an interface which provides a Subsystem port. The boards and the host are connected together by connecting the host Subsystem port to the Up port of Transputer board 1, and the Down port on each but the last transputer board is connected to the Up port on the next board. This daisy-chains the subsystems on the boards and allows the host to reset or analyse all the transputers with a single output signal and detect an error on any transputer by polling a single input signal.

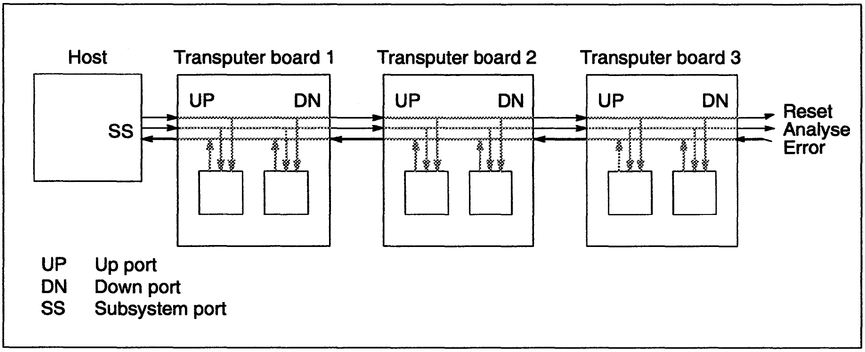


Figure 8.1 Example system services wiring

8.4.2 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 multi-transputer system, the Up port of the root transputer board is connected to the host computer, as in figure 8.1, so that the host can control the loading of programs and monitor errors on the network.

In a simple application requiring multiple transputers, the subnetwork would normally be connected to Down on the root transputer board. 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. If the INQUEST debugger is to be used for post-mortem debugging, then the transputer boards must be connected in this way, as in figure 8.1, so that the debugger running on the host can detect errors from any transputer in the network.

If the next processor after to root is the first processor in a target subnetwork, it is normally connected to either Down or Subsystem depending on the application, and other boards in the target subnetwork are chained together via their Up and Down ports.

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.

8.5 AServer and the AServer database

8.5.1 AServer

The AServer (Asynchronous Server) system is an interface system which allows multiple processes on a target device to communicate via a hardware serial link with multiple processes on some external device. The AServer software acts as a standard interface which is independent of the hardware used.

The AServer is a collection of programs, interface libraries and protocols that together create a system to enable applications running on target hardware to access external services in a way that is consistent, extensible and open. `irun` is an AServer multiplexing process, called a gateway, which runs on the host and handles the hardware interface. The AServer is described in more detail in the *AServer Programmers' Guide.*

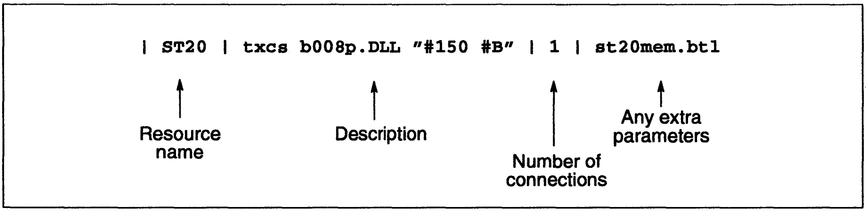
8.5.2 AServer database

`irun` 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 `irun` command line or in the environment variable `TRANSPUTER`. `irun` gets information about the specified User Link from an AServer database file.

The AServer database is a text file. It may be called `aservdb` and be on the `ISEARCH` path or it may be pointed to by the `ASERVDB` environment variable. It consists of a list of resources. Each resource is either:

- a target hardware connection which may be accessed by the host or

- a host software process which may be requested by the application, including the debugger. The use of these processes for customizing the host interface is described in the *AServer Programmer's Guide.*



```
| ST20 | txcs b008p.DLL "#150 #B" | 1 | st20mem.btl
```

Figure 8.2 AServer database line

Each resource is represented by one line of the database file. Each line contains four fields, each preceded by the vertical bar character (|), as shown in figure 8.2. The entries are case sensitive, so, for example, `ST20` is not the same as `st20`.

The AServer database is described in more detail in an appendix to the *Toolset Reference Manual.*

8.6 Skip loading

This section describes how to skip load a program onto a subnetwork, skipping over one or more transputers, using the skip bootables. The skip bootables are described in more detail in the `irun` chapter of the *Toolset Reference Manual.*

The skip bootables allow a program to be loaded onto a transputer network which has no direct link connection to the host, but is connected via one or more intermediate transputers. Each skip bootable starts up a special route-through process on the transputer that transfers data between the rest of the network and the host system.

There are four skip bootables, `skip0.btl`, `skip1.btl`, `skip2.btl`, and `skip3.btl`, one for each link. When a skip bootable is loaded onto a transputer, it boots the transputer and runs code to pass on all messages, making the transputer effectively transparent to the software. The link from which the transputer is booted is called the boot link. If the skip bootable `skip`*n*`.btl` is used then all messages arriving on the boot link are passed on to link *n* and all messages arriving on link *n* are passed to the boot link. This means that subsequent code loaded onto the network is passed to the transputer connected to link *n*, so that transputer becomes the new root transputer. Host communications from the new root transputer are passed back to the host. Any number of skip bootables may be loaded to skip over that number of transputers.

The skipped transputers must not appear in the configuration. The root transputer in the configuration must be the first transputer after skipping over the skipped transputers.

For example, in figure 8.3 there is one transputer between the configured target network and the host. The target network is connected to link 2 of the skipped transputer. The skipped transputer will run `skip2.btl`. The skip bootable `skip2.btl` is loaded first onto the skipped transputer and then the user application is loaded onto the network beyond.

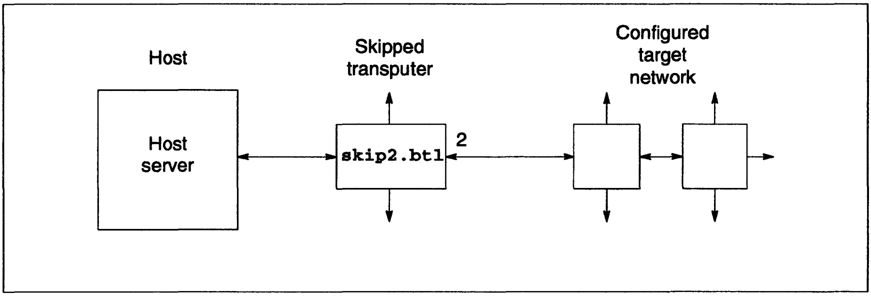


Figure 8.3 Skipping over one transputer

In figure 8.4 there are two transputers to be skipped between the configured target network and the host. The target network is connected to link 3 of the second skipped transputer, which runs `skip3.btl`. The second skipped transputer is connected to link 1 of the first skipped transputer, which runs `skip1.btl`. The skip bootable `skip1.btl` is loaded first onto the first skipped transputer, then `skip3.btl` is loaded onto the second skipped transputer and then the user application is loaded onto the network beyond.

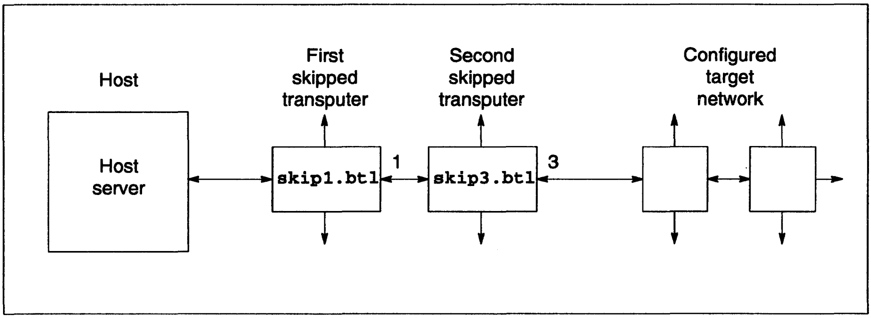


Figure 8.4 Skipping over two transputers

8.6.1 Invoking skip bootables

Skip bootables are normally specified in the AServer database. This method is preferred as it allows post-mortem debugging. The AServer database is described in more detail in an appendix of the *Toolset Reference Manual.*

A separate resource may be defined to connect to each appropriate subnetwork, skipping over one or more transputers. The skip bootables must be given in the extra parameters field preceded by `-a`, which tells `irun` to reload the skip bootable when the network is analyzed for post-mortem debugging.

For example, the following is an AServer database line to define a User Link called `TptrSkip`. The target network is connected to link 2 of the root transputer, as in figure 8.3, where the root transputer is on an IMS B008 board:

```
| TptrSkip | txcs b008p.DLL "#150 #B" | 1 | -a skip2.btl
```

If there are several transputers to skip over, then each skip must be preceded by `-a`, and the order of the skip bootables should be the same as the order of the transputers being skipped.

For example, for skipping over two transputers, as in figure 8.4:

```
| DblSkip | txcs b008p.DLL "#150 #B" | 1 | -a skip1.btl -a skip3.btl
```

A skip loading bootable may also be invoked by adding the skip bootable as a SC option on the `irun` command line. For example, to load *bootable_file* on the network connected to link *n* of the root transputer, use the command:

```
irun -sc skipn.btl bootable_file
```

The following command would be used to load and run the application `example.btl` on the network shown in figure 8.3:

```
irun -sc skip2.btl example.btl
```

This command resets the network, loads the skip loader onto the root processor, sends the bootable file `example.btl` and serves the link, i.e. responds to requests from the `example.btl` application. This has the effect of running `example.btl` on the transputer network found down link 2 from the skipped transputer. The `-sc skip2.btl` option sets up link 2 as the path to the target network and starts the route-through process on the skipped transputer. The server then loads `example.btl` without resetting the network.

The following command would be used to load and run the application `example.btl` on the network shown in figure 8.4.

```
irun -sc skip1.btl -sc skip3.btl example.btl
```

This command loads `skip1.btl` then `skip3.btl` then loads `example.btl` onto the network beyond. The order of the skip bootables is significant.

8.7 Clearing error flags

Error flags can become set when a transputer board is powered up, since error flags are not cleared by resetting. Error flags are normally cleared by the bootstrap code. If an application does not load code onto all the transputers in a network then one or more error flags may persist after the application is loaded. This may be immediately detected by the host server, which will halt and display the message `Transputer error flag set`.

To avoid this happening, error flags can be cleared by using a network analyzer program such as `rspy` prior to loading the application. The `rspy` program is provided as an INQUEST tool and is described in the *INQUEST User and Reference Manual.*

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.

9 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.

9.1 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 `irun` tool supplied with this toolset. Programs that use the i/o libraries should always be loaded using `irun`.

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 6.

9.2 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.

9.2.1 The host file server

The host file server provides the runtime environment that enables application programs to communicate with the host. It contains support 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;

9.2.2 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 i/o

All routines in these libraries are independent of the host operating system. They offer same access to the host file server, but using a different model.

The `hostio` library contains basic routines for accessing files and controlling the file system. The `hostio` library routines are closely related to the functionality provided by the host server and provide efficient access to the host services. Each routine requires access to a pair of channels to and from the server using the protocol `SP`. Generally each host access requires a completed hand-shake.

The `streamio` library provides i/o support with a higher level of abstraction, called data streams, on top of the basic i/o provided by the `hostio` library. Different protocols `KS` and `SS` are used for keyboard and screen streams respectively. Each stream uses a single channel without handshaking, which simplifies and speeds up buffering and multiplexing. However, these streams are not directly supported by the host server, so interface process routines are provided which convert between the streams and the server protocol.

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 2.1 Toolset Language and Libraries Reference Manual.*

9.2.3 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 host file 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`.

9.3 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 configured programs the channels are declared in the configuration and 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. For unconfigured programs the channels are defined as formal parameters to the program.

9.4 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 9.1.

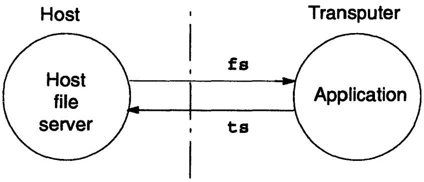


Figure 9.1 Program input/output

If more than one process requires access to the host then the server may 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. An alternative solution is to use the AServer's multiplexing capability or multiple host services. These facilities are described in the *AServer Programmers' Guide*.

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.

9.4 Multiplexing processes to the host

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 chapter 4 of the *occam 2.1 Toolset Language and Libraries Reference Manual.*

An example of a multiplexed system is shown in figure 9.2, and the occam code that would implement the system is as follows:

```
#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 server
          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)
:
```

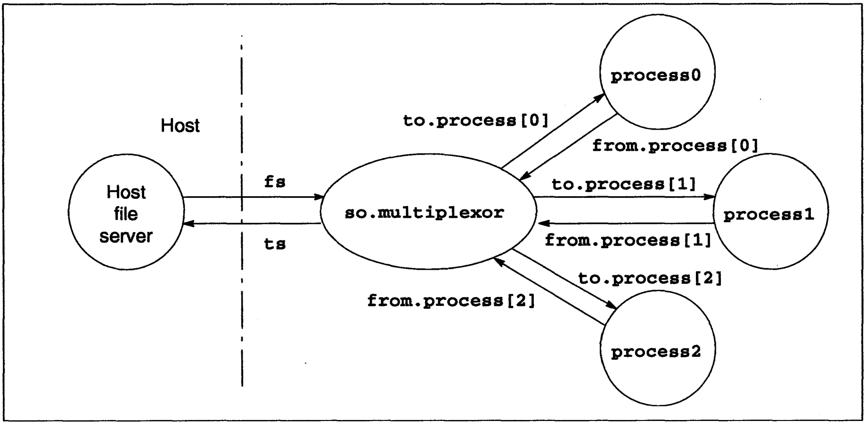


Figure 9.2 Multiplexing access to the host file server

Multiplexor processes can be chained together to produce any degree of multiplexing to the host. However, the host is a single, finite resource and unrestrained multiplexing of processes should be avoided if possible.

9.4.1 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.

9.4.2 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.

Part 2: Advanced techniques

10 Advanced use of the configurer

This chapter describes the advanced use of `occonf` and is aimed at users who wish to override certain configuration defaults. The chapter deals with the following topics:

- INQUEST support;
- Memory usage by the configurer;
- Channel communications.

The chapter describes how to override the default allocation of user's code and data in memory and how 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 Support for INQUEST

Two attributes are included for use with the INQUEST debugger and profilers, 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 processor, that processor will be ignored.

The default value of the `nodebug` attributes is `FALSE` when the configurer `GA` option is used and `TRUE` when the `GD` option is used. If the `GD` option is used, then for a processor to be debugged its `nodebug` attribute must be set to `FALSE`. The default value of the `noprofile` attribute is `FALSE`.

10.2 Code and data placement

The configuration language provides `reserved`, `location` and `order` processor attributes for influencing the use of memory. These attributes are described in sections 10.2.3, 10.2.4 and 10.2.5 respectively. This section describes the circumstances in which these attributes should be used.

The `location` and `order` attributes are normally disabled and must be explicitly enabled by the configurer option `RE`.

10.2.1 Default memory map

By default the configurer maps code and data into memory in the following order beginning at **LoadStart**: workspace; code; vector space and system data. The code and data segments are contiguous. The upper limit of the memory available to the configurer is defined by the `memsize` attribute specified for the processor. For details of the default memory map and a definition of **LoadStart**, see chapter 4 of the *Toolset Reference Manual*.

10.2.2 Other memory configurations

Figure 10.1 illustrates a memory configuration with additional requirements to those provided by the configurer in the default case. To cater for such situations the `reserved` and `location` attributes are supported by the configuration language.

Figure 10.1 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, as given by the default memory map.

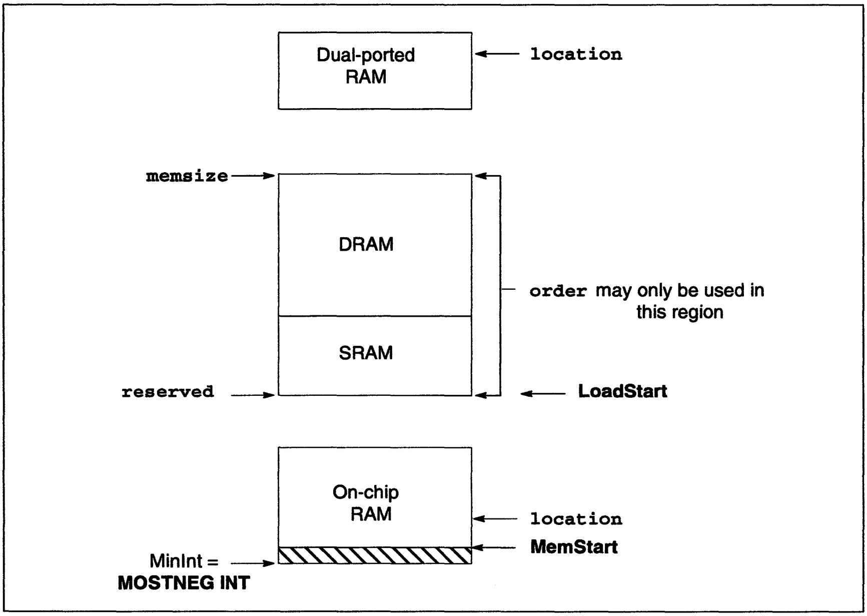


Figure 10.1 Example discontinuous memory map

10.2.3 Reserving memory

A region of memory may be reserved using the processor attribute `reserved`. The region is specified as a number of bytes starting at the base of memory, `MOSTNEG INT`. Since this attribute is a property of the 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.

It will be possible for the user, using the `location` attributes to place the code and data segments of user processes into the reserved memory.

By default, the configurer uses memory in a contiguous block from the bottom of the transputer's available memory at **LoadStart** to the top of the memory specified by the `memsize` attribute. This can be overridden by means of the `reserved` attribute which specifies the number of bytes of memory which should be reserved from `MOSTNEG INT`. This reserved memory cannot be used by the configurer to place user or system processes. The `reserved` attribute must be set to a positive value.

For example:

```
MAPPING reserve.low.memory
  DO
    MAP logical ONTO physical
    SET physical (reserved := 4*K)
:
```

This would ensure that the bottom 4 Kbytes of memory are reserved and will not automatically be used by the configurer.

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; see section 4.15.1 of the *Toolset Reference Manual.*

In figure 10.1 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.

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 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 from the default **LoadStart** offset for the processor to the top of memory (as specified by the `memsize` attribute).

10.2.4 Absolute address code placement

The `location` processor attributes allow the code and data segments of a processor to be placed at absolute addresses in the transputer's address space. Since these attributes are properties of the 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 addresses referenced by the `location` attributes can only lie in the region of memory reserved by the `reserved` attribute, or in a region of memory above `memsize` bytes from the bottom of memory.

If the `location` attributes are not specified then the configurer will automatically place non-addressed code and data segments.

The location attributes are `location.code`, `location.ws` and `location.vs`, and are defined in table 7.4.

For example:

```
MAPPING use.absolute.addresses
  DO
    MAP logical ONTO physical
    SET physical (reserved    := 4*K)
    SET physical (location.ws := #80000100)
:
```

This would ensure that the bottom 4 Kbytes of memory are reserved and will not automatically be used by the configurer, except that the workspace is placed at address `#800000100`.

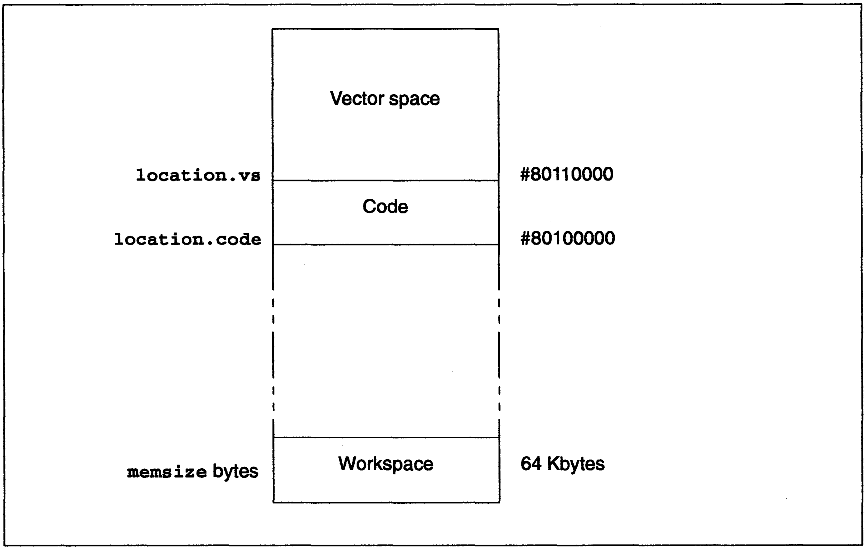


Figure 10.2 Example use of location attributes

In the following example, the `memsize` of the processor has been set to 64 Kbytes but the transputer has additional memory starting at address #80100000, as shown in figure 10.2:

```
MAPPING use.absolute.addresses
  DO
    MAP logical ONTO physical
    SET physical (location.code := #80100000)
    SET physical (location.vs   := #80110000)
:
```

This would place the code at address #801000000, and the vector space at #80110000.

The `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. The `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 the:

- 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 4.15.1 of the *Toolset Reference Manual* for more information about **LoadStart**).
- 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 received 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 user'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**.

Figure 10.1 indicates how the `location` attributes can be used to access memory below **LoadStart** (which has been changed from its default value by the `reserved` attribute) or spare memory locations available in external RAM.

10.2.5 Ordering code and data segments

The `order` processor attributes may be used to provide greater control over the layout of code and data segments of a processor in memory. Since these attributes are properties of the 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 named in the hardware description.

By default, the code, workspace and vector space all have the same priority. If these regions have the same priority then the configurer arranges for the program's workspace to be placed at the lowest addresses in memory. This means that the workspace can make best use of the transputer's on-chip RAM. Program code is placed immediately above the workspace, and vector space is placed above the code.

The default ordering can be overridden by setting one or more of the three processor attributes `order.code`, `order.ws` and `order.vs`. These attributes set the priorities of the code, the workspace, and the vector space of the program respectively. The priorities are by default all set to 0.

An `order` attribute can be set to any `INT` value, where a lower value means a higher priority and hence placement at lower addresses. Therefore setting `order.code` to –1 means that the code on the processor will be placed at a lower address than the workspace or vector space.

Thus we may have a mapping section like the following:

```
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 vector space has no priority defined and is therefore placed by default after the workspace.

The `order` attributes must be enabled on the configurer command line by using the `RE` option. If this option is not specified these attributes will be ignored.

The `order` processor attributes can still be used in conjunction with the `reserved` and `location` attributes. The `order` attributes are used to change the ordering priority of those processor 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.

If both the `location` and `order` attributes are specified for a particular segment then the `location` attribute will override the `order` attribute.

10.3 Channel communication – configuration techniques

When software virtual routing is required, the configurer works by adding multiplexing and de-multiplexing 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 link 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.
- It is possible to prevent specific processors from being used as routes 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` (or `MAP`) statement and the `order` attributes to optimize important channels and to make the best use of fast memory. Section 10.4 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 included in section 10.5.

10.3.1 Routing and placement constants

The include file `occonf.inc` contains a number of constants associated with the routing and placement attributes. The file must 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. The statement to reference the file is:

```
#INCLUDE "occonf.inc"
```

10.3.2 Optimizing important application channels

By placing an application channel on an `ARC` it is possible to reserve the hardware link solely for the use of the application channel concerned.

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 channels.

When making placements 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 10.4.4.

10.3.3 Virtual communications – use of fast memory

Normally the workspace of virtual channel support processes (added to the target network by the configurer) 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 workspace of the virtual channel support processes is allocated. This is done by setting `order` attributes for the relevant processor to lower values than those automatically given to the workspace segments of the virtual channel support processes.

Virtual channel support processes are divided into *routing* processes and *multiplexing/ demultiplexing* processes. The workspace segments of the routing processes placed by the configurer are all given the `order` value `ROUTER.ORDER`. The workspace segments of multiplexing/demultiplexing processes placed by the configurer are all given the `order` value `MUXER.ORDER`. Values of `-20000` and `-10000` respectively are defined for these constants in the include file `occonf.inc` which is supplied with the toolset.

If the `order` values of the code and data segments of the processor are less than `ROUTER_ORDER` then the ordered segments will be allocated before any of the virtual channel support processes' workspaces are allocated.

If the `order` values of the code and data segments of the processor are less than `MUXER_ORDER` but greater than `ROUTER_ORDER`, then only the workspaces required by routing processes will be allocated before the configurer allocates space for the ordered segments.

If the workspaces 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.

10.4 Control of routing

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.

The routecost, tolerance, and linkquota processor attributes are used to control the way that virtual routing is performed. Since these attributes are properties of the 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.

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

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

The exact behavior of these attributes is defined in the following sections.

10.4.1 Routing cost

The routecost attribute 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.

The routecost attribute 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 processor 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.

10.4.2 Tolerance

The tolerance attribute 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.

10.4.3 Link quota

The `linkquota` attribute 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 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.

10.4.4 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 10.3.2) it is possible to `PLACE` (or `MAP`) 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 are 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*.

10.4.5 Summary of routing attributes

The routing attributes may be summarized as follows:

- `routecost` - defines within the range `MIN.COST` = 1 to `MAX.COST` = 1000000 inclusive, the associated cost of routing virtual channels through a particular processor.

  If a value greater than `MAX.COST` is specified (e.g. `INFINITE.COST`, which has the value `MAX.COST`+1) then no through-routing will be permitted on that processor.

  If this attribute is not defined for a particular processor then the cost value `DEFAULT.COST` = 1000 will be assumed.

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

- `tolerance` - controls with any value in the range `ZERO.TOLERANCE` = 0 to `MAX.TOLERANCE` = 1000000 inclusive, how much a particular processor can be used to provide load-sharing routing paths for other processors.

  The default value for this attribute is `DEFAULT.TOLERANCE` = 1. 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 `tolerance` is set to `MAX.TOLERANCE` on all processors in the target network all possible routes will be used to share the cost of carrying data between any pair of non-adjacent processors.

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

- `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 processor is exceeded. The `linkquota` will only be exceeded because of the requirements of through-routing data for other processors.

10.4.6 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 `DEFAULT.COST`. 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.

10.4.7 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, e.g. `MIN.COST`, 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.

10.4.8 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 approximate figures for the minimum and maximum costs of both through-routing and multiplexing software on different word length transputers are shown table 10.1.

Word Size	Function	Code	Min workspace	Max workspace
32 bits	Through-routing	720	760	2120
	Multiplexing	2100	780	2060
16 bits	Through-routing	720	510	1570
	Multiplexing	2100	520	1560

Table 10.1 Virtual routing approximate memory usage in bytes

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.

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 case the processor can be completely protected from run-time overheads by using the techniques described in section 10.4.6.

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.

10.5 Example – optimized filter test program

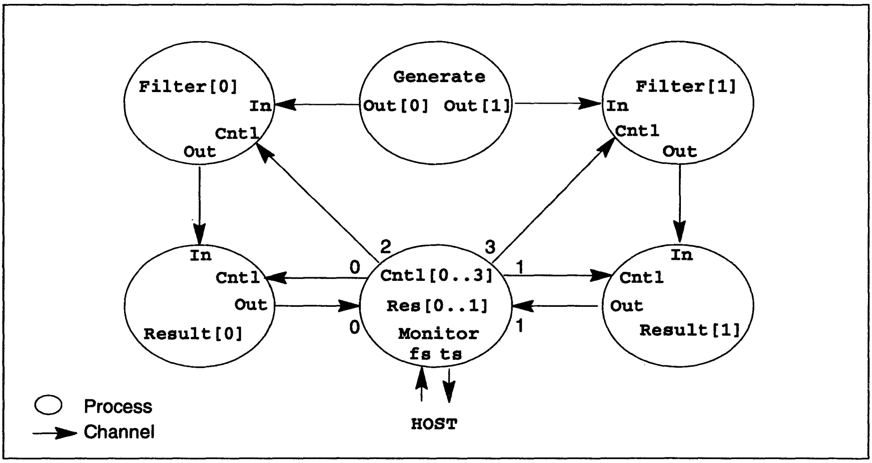


Figure 10.3 Example filter test program

Figure 10.3 shows an example software process network that needs to be placed onto a network of six processors, shown in 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.

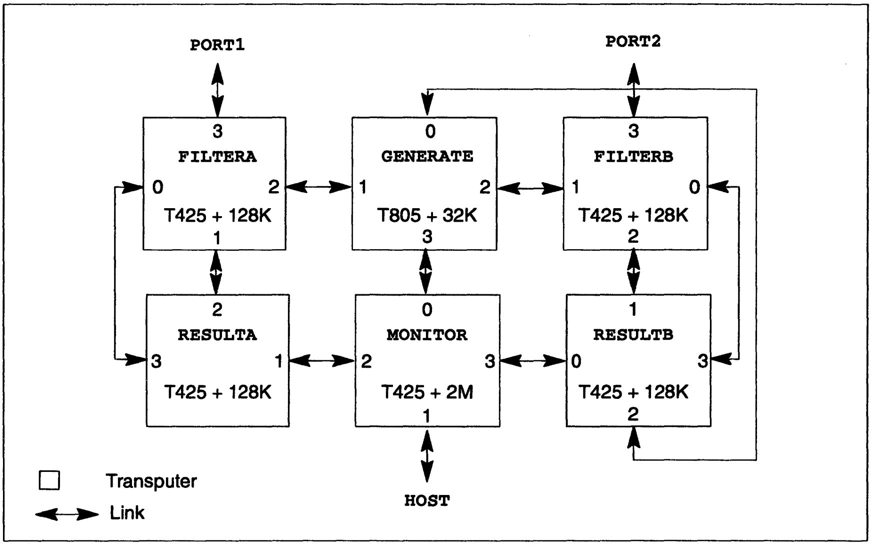


Figure 10.4 Example filter test hardware

10.5 Example – optimized filter test program

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. It is similar to an example supplied in the `examples` directory. The basic configuration description is as follows:

```
-- 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 := "T805",  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.1, monitor.1 :
[2]NODE result.1, filter.1 :

#INCLUDE "hostio.inc"
#USE "generate.c8h"
#USE "filter.c5h"
#USE "result.c5h"
#USE "monitor.c5h"
```

```
CHAN OF SP fs, ts :
[2]CHAN OF BYTE Generate.to.Filter :
[2]CHAN OF BYTE Filter.to.Result :
CONFIG
  [2]CHAN OF BYTE Result.to.Monitor :
  [2]CHAN OF BYTE Monitor.to.Result, Monitor.to.Filter :
  PAR
    PROCESSOR monitor.1
      Monitor (fs, ts, Result.to.Monitor,
                       Monitor.to.Result,
                       Monitor.to.Filter)
    PROCESSOR generate.1
      Generate (Generate.to.Filter)
    PAR i = 0 FOR 2
      PAR
        PROCESSOR result.1[i]
          Result (Filter.to.Result[i],
                  Result.to.Monitor[i],
                  Monitor.to.Result[i])
        PROCESSOR filter.1[i]
          Filter (Generate.to.Filter[i],
                  Filter.to.Result[i],
                  Monitor.to.Filter[i])
:

-- Mapping description
MAPPING
  DO
    MAP generate.1  ONTO GENERATE
    MAP filter.1[0] ONTO FILTERA
    MAP filter.1[1] ONTO FILTERB
    MAP result.1[0] ONTO RESULTA
    MAP result.1[1] ONTO RESULTB
    MAP monitor.1   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 Generate.to.Filter[0] ONTO GENERATE.TO.FILTERA


    -- Optimise Generate to Filter 1 Path
    MAP Generate.to.Filter[1] ONTO GENERATE.TO.FILTERB

    -- Optimise Filter to Result 0 Path
    MAP Filter.to.Result[0] ONTO FILTERA.TO.RESULTA

    -- Optimise Filter to Result 1 Path
    MAP Filter.to.Result[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 routecost to the value 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 Generate.to.Filter and Filter.to.Result channels must both be placed onto hardware links to guarantee the required performance. The Monitor.to.Result and Monitor.to.Filter channels which carry a small amount of parameterization data can, however, be implemented as virtual channels without significant 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 (required by C) 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. Non-VAL parameters are always passed as pointers.

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 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

occam 2.1 RECORD types cannot be passed to C, and C cannot call an occam function that accepts a RECORD as a parameter. Similarly, C struct types cannot be passed to occam and occam cannot call a C function that accepts a struct as a parameter. Named occam data types and C typedef types may be passable by using the rule which applies to the underlying data type from which they are derived.

Comprehensive equivalence tables, with examples of calling external routines from each language, can be found in Appendix A.

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);
```

(The pragma used is described below in section 11.1.6; all pragmas supported by the toolset are described in the *Toolset Reference Manual*).

11.1.4 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 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 three 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 ANSI 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 described in the *ANSI C Toolset User Guide*

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 A.

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

C cannot call an occam function that returns a RECORD, and occam cannot call a C function that returns a struct.

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"
⋮

INT y:
SEQ
  y := 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 cannot call an occam function that accepts a RECORD as a parameter or returns a RECORD.

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 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 11.1.4). 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 11.1.4). 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.

11.1.7 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 `occamx.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 `occamx.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` (supplied with the ANSI C Toolset) 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 implicitly, 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 11.1.4.

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.

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 of memory 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 `terminate.heap.use` 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 `terminate.static.use` 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.

The linker allows files to either be specified on the command line or listed in an indirect file. Because there are several files required in this instance, it may be easier to supply a linker indirect file. To do this create a text file called `callc.lnk`, containing the following lines:

```
mixed.tco
cfunc.tco
#include clibsrd.lnk
hostio.lib
callc.lib
#include occama.lnk
```

The occam module `mixed.tco` is listed first, because it contains the main entry point of the program. Alternatively a `#mainentry` directive could be used to define the main entry point.

```
ilink -t425 -f callc.lnk
```

Details of the operation of the linker can be found in chapter 10 in the *Toolset Reference Manual*.

Once linked, the program can be configured and 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* 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;
- perform heap operations.

4 The following functions cannot be called in the imported C code:

`clock()`

`exit()` (or any of its variants)

`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
```

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, 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.

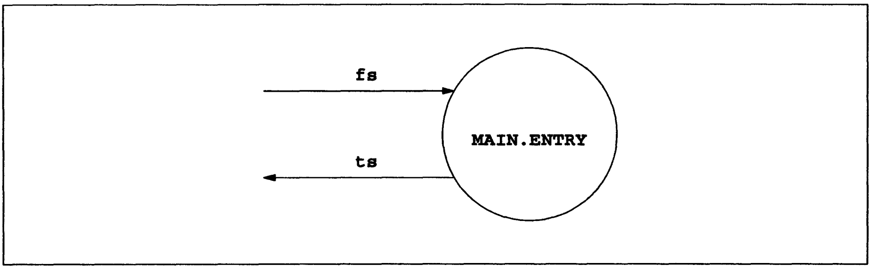


Figure 11.1 Type 1 interface

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.

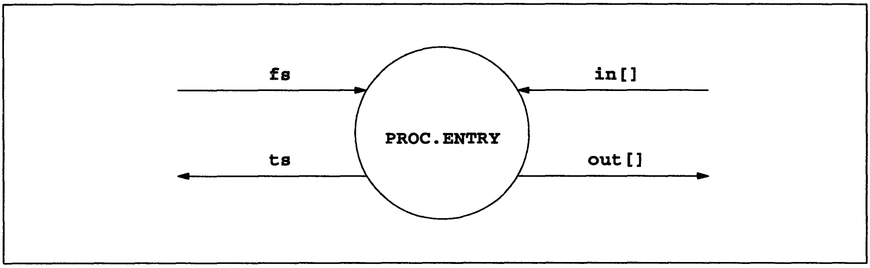


Figure 11.2 Type 2 interface

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 the host.

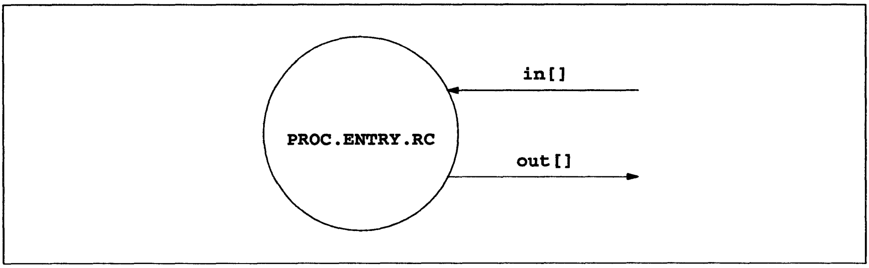


Figure 11.3 Type 3 interface

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 11.2.2).

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 2.1 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.
`in[0]`	Reserved.
`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.

This interface to main is only valid when using the type 2 and type 3 occam interface procedures. This interface to main is *not* valid in any other situation.

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 the chapter 15 in this manual.

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 INQUEST 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

  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 all 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 T425 transputer, using the following command:

```
ilink wrap.tco main.tco callc.lib -t425 -f clibs.lnk -f occama.lnk
```

12 Low level programming

This chapter describes a number of features of the toolset occam 2.1 compiler which support low-level programming of transputers. These are as follows:

Allocation to memory 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 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 T2/T4/T8-series transputer error flag can be explicitly set or an ST20450 trap raised using the predefined routine CAUSEERROR.

12.1 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 2.1 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
```

12.1.1 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 2.1 Toolset Language and Libraries Reference Manual*, and cannot allow data to be placed arbitrarily in memory. To make the best use of fast RAM use separate vector space as described in section 6.5.

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 12.1.3.

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.

12.1.2 Allocating specific workspace locations

A number of specialized transputer instructions require specific workspace placements. For example, the instructions POSTNORMSN, OUTBYTE, OUTWORD and the disabling ALT instructions all use workspace location 0. When programming in occam this is taken care of by the compiler, however, if these instructions are used within an ASM construct, the programmer must specifically reserve the location. See section 12.3.4. To accommodate this the compiler supports the following allocation:

PLACE *name* AT WORKSPACE *expression*:

where: *expression* is a constant of type INT; see Appendix A in the *occam 2.1 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 a sufficient number of 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 an IMS 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 12.3.

12.1.3 Allocating channels to links

The facility to map channels onto links from the source code is largely redundant:

- it is not recommended that channels between processors are allocated to specific links as the configurer will perform the allocation automatically;

- the configuration language enables channels to be explicitly mapped to links for those situations where this is mandatory i.e. for channels at the edges of a network e.g. connecting the host or a peripheral device to the network.
- If a link is described in a configuration description then it should not be used directly as well. This is because all links known to the configurer will be used by it when forming the network connections. In addition, the configurer may use any of the links in order to set up virtual links. Attempting to use any links directly in such situations will result in undefined behavior.

The mechanism for allocating channels to links from the source code is described here for completeness, however, it is strongly recommended that only mandatory allocations are performed and that they are done using the configuration language; see chapter 7.

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.

A T2/T4/T8-series transputer only has the one event pin. Event addresses are defined in the include file **linkaddr.inc** that is supplied with the toolset.

The following code fragments illustrate the allocation and use of event channels.

```
CHAN OF BYTE  event :
PLACE   event  AT event.in:

BYTE dummy:
event ? dummy
```

Although the example shown here use CHAN OF BYTE channels you could use any channel protocol.

12.2 Retyping channels and creating channel array constructors

Channels may be RETYPEd (see also A.2 of the *occam 2.1 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 RETYPEing channels should only be used by programmers who understand the implementation of transputer channels, and the implications of attempting to circumvent the checking of channel usage in occam. 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 P 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 P 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 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, for example:

```
[n]INT x:
SEQ
  ... initialize elements of array x to MOSTNEG INT

  [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 of the occam *2.1 Toolset Language and Libraries Reference Manual*). Channel array constructors may be used for this purpose:

```
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
:
```

12.3 Code insertion

This section describes the facilities provided by the occam 2.1 compiler code insertion mechanism using ASM. A formal description is given in Appendix B.

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 T2/T4/T8-series transputer instruction set may be found in *The transputer instruction set: a compiler writer's guide*. The ST20450 instruction set is described in *The ST20 Instruction Set Reference Manual.*

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.

12.3.1 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 B.

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 B.3.

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 directives recognized by the compiler. They expand into one or more instructions, depending upon their context and parameters.

For example, to perform a 1's complement addition we can write the following:

```
INT carry, temp:
SEQ
  carry, temp := LONGSUM (a, b, 0)
  c := carry PLUS temp
```

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 (T2/T4/T8-series transputers) or traps (ST20450) 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
```

12.3.2 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.

12.3.3 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
```

Labels are declared in a different name space 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.

12.3.4 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 12.1.2 for details of workspace allocation.

12.3.5 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 instructions listed in table 12.1 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

Table 12.1 Instructions requiring workspace slots

12.3.6 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.

12.3.7 Programming notes

1 Floating-point (fp) registers cannot be loaded directly; they must be loaded or stored by first loading a pointer to the value to be loaded, into an integer register and then using the appropriate floating-point indirect load 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.

12.4 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 FUNCTIONs, it is recommended that you call the FUNCTION indirectly from an occam PROC, and use non-VAL parameters to return the results to the calling environment).

The procedures for setting up parameters before the call and for making the call are outlined in the table below, and described in the following sections, with examples. Further information and examples of this technique can be found in section 5.3.5 of *The Transputer Applications Notebook – Systems and Performance*.

Procedure	Parameter Specifiers

`KERNEL.RUN`	`VAL []BYTE code,`
	`VAL INT entry.offset,`
	`[]INT workspace,`
	`VAL INT no.of.parameters`

`LOAD.INPUT.CHANNEL`	`INT here, CHAN OF ANY in`
`LOAD.INPUT.CHANNEL.VECTOR`	`INT here, []CHAN OF ANY in`
`LOAD.OUTPUT.CHANNEL`	`INT here, CHAN OF ANY out`
`LOAD.OUTPUT.CHANNEL.VECTOR`	`INT here, []CHAN OF ANY out`
`LOAD.BYTE.VECTOR`	`INT here, VAL []BYTE bytes`

12.4 Dynamic code loading

The collector tool `icollect` can produce code in a format suitable for dynamic loading. The tool is described in chapter 5 of the *Toolset Reference Manual.*

12.4.1 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 12.1. 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.

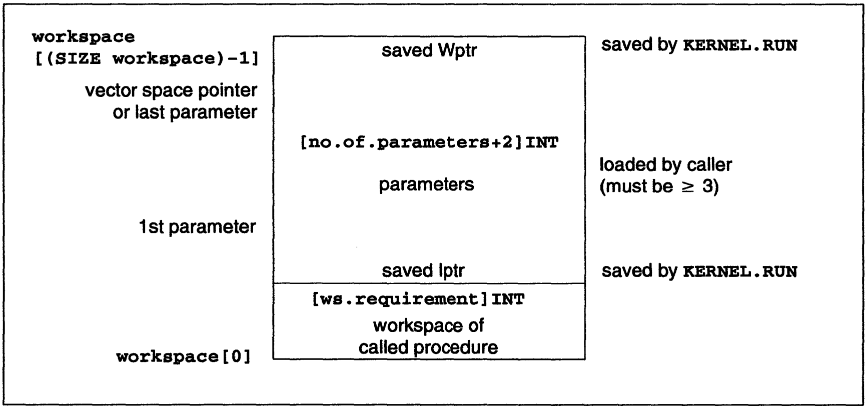


Figure 12.1 Workspace buffer

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; 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.

12.4.2 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 multi-dimensional arrays, one parameter is needed for each unsized dimension, in the order that the dimensions are declared.

All variables and arrays should be retyped to byte vectors before using `LOAD.BYTE.VECTOR` to obtain their addresses, using a retype of the form:

```
[]BYTE b.vector RETYPES variable:
```

`LOAD.BYTE.VECTOR` should also be used to set up the address of the separate vector space.

12.4.3 Examples

This section gives two examples of dynamic loading. The first is a simple example showing how code without parameters 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` directory.

Example 1: load from link and run

This is a simple procedure to load a code packet without parameters from a link and run it. The type of the packet is given by the protocol:

```
PROTOCOL CODE.MESSAGE IS INT::[]BYTE; INT; INT
```

The code is sent first, as a counted array, followed by the entry offset and workspace size.

```
PROC run.code (CHAN OF CODE.MESSAGE input,
               []INT run.vector, []BYTE code.buffer)
  VAL no.parameters IS 3 :   -- smallest allowed
  INT code.length,entry.offset,work.space.size:
  INT total.work.space.size:
  SEQ
    input ? code.length::code.buffer;
             entry.offset; work.space.size
    total.work.space.size :=
       work.space.size + (no.parameters + 2)
    []INT  work.space IS [run.vector FROM 0 FOR
                           total.work.space.size] :
    KERNEL.RUN (code.buffer, entry.offset,
                work.space, no.parameters)
:
```

Example 2: system loader

This example shows how to set up parameters prior to running code loaded from a file. It is assumed that the code requires use of a separate vector space.

Consider a process with an entry of the form:

```
PROC process (CHAN OF 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 5 in the *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 12.2.

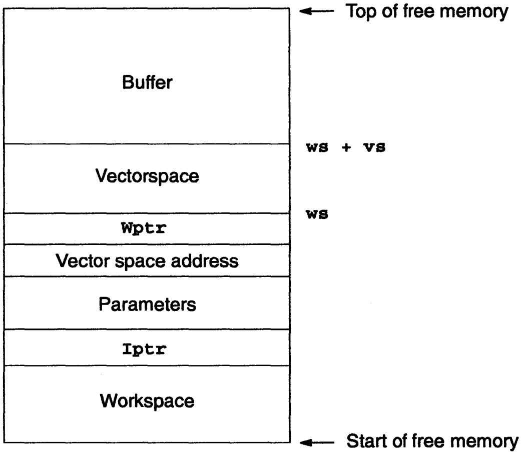


Figure 12.2 Partitioning of free memory

12.5 Extraordinary use of links

The transputer link architecture provides synchronized communication at the message level which matches the occam model of communication.

In certain circumstances, it is desirable to use a transputer link even though the synchronized message passing of occam is not exactly what is required. For example, if a process cannot be certain that a processor exists or that the hardware is correctly connected, then a trial output may be attempted. Using normal occam communication, it is difficult to recover from a failure of such a trial. In such cases, it would be desirable to be able to abort a link communication before it is completed. Such extraordinary use of transputer links is possible but requires great care and the use of some special occam procedures.

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.

The use of the extraordinary link procedures is described in this chapter. To use them in a compilation unit, the directive `#USE "xlink.lib"` should be inserted at the beginning of the source for that unit. For details of the procedures see chapter 9 in the *occam 2.1 Toolset Language and Libraries Reference Manual.*

The extraordinary link procedures provided are necessarily slow, and should not be used routinely where performance is important. In many cases it is sufficient to use these procedures once to test that the link is connected and an executing process is available at the other end to communicate. Once this has been established, normal channel i/o should be used.

The extraordinary link procedures may only be used with link channels which are direct, as defined in section 7.5.2, and may not be used with interactive debugging.

12.5.1 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 re-initializing the link hardware; again there is a suitable library procedure to allow this to be performed.

12.5.2 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 are intended to be used as input/output pairs in order to provide secure communication at both ends of the link; failure to do this could lead to undefined results. Thus `InputOrFail.t` should be paired with `OutputOrFail.t` and the procedure `InputOrFail.c` should be paired with `OutputOrFail.c`, see below.

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 12.2) 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)
```

Note: these procedures must not be used on virtual channels implemented in software.

12.5.3 Recovery from failure

To reuse a link after a communication failure has occurred it is necessary to reinitialize the link hardware. This involves re-initializing both ends of both channels implemented by the link. Furthermore, the re-initialization must be done after all processes have stopped trying to communicate on the link. So, although the `InputOrFail` and the `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.

12.5.4 Example – unreliable connections

For our example consider the network illustrated in figure 12.3. The master transputer needs to be able to recover from a link connection failure.

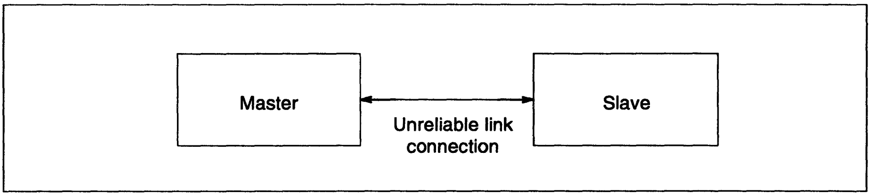


Figure 12.3 Unreliable connection

The first step in the solution is to recognize that the master knows when a failure might occur, and hence knows when it might be necessary to abort a communication.

When the master decides to reset the slave it can send a message to the interface process directing it to abort any transfers in progress. It can then reset the slave (which resets the slave end of the link) and reinitialize the link.

The example program below could be that part of the master code which runs when the slave starts executing and continues until the slave is reset and the link is re-initialized.

```
SEQ
  CHAN OF SIGNAL end.input, end.output :
  PAR
    interface (link.in, link.out, end.input, end.output)
    monitor (end.input, end.output)
  ...  reset slave
  Reinitialise(link.in)
  Reinitialise(link.out)
```

The monitor process will output on `end.input` and `end.output` when it detects an error on the slave.

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
      end.output ? any
        active := FALSE
      from.main ? message
        link.out ! message
    ...
```

This process will loop, forwarding input from `from.main` to `link.out`, until it receives a message on `end.output`. However, if the slave 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
      end.output ? any
        active := FALSE
      from.main ? message
        SEQ
          OutputOrFail.c (link.out, message,
                          end.output, aborted)
          active := NOT aborted
```

This program is always prepared to input from `end.output`, and is always terminated by an input from `end.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.

12.6 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.

12.7 Setting the error flag

For T2/T4/T8-series transputers (including ST20450) the transputer error flag can be explicitly set 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.

On a T2/T4/T8-series transputer, 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.

13 EPROM programming

The toolset EPROM software is designed so that programs can be developed, booted onto a network via link and tested without using ROMs. Once they are working, they can be placed in ROM with only minor changes.

13.1 Introduction

During development, software is booted onto a network from a link connecting the network to the host computer. The software is then prepared for a ROM, which is attached to the root transputer in the network.

Figure 13.1 shows how a network of five transputers would be loaded from a ROM accessed by the root transputer.

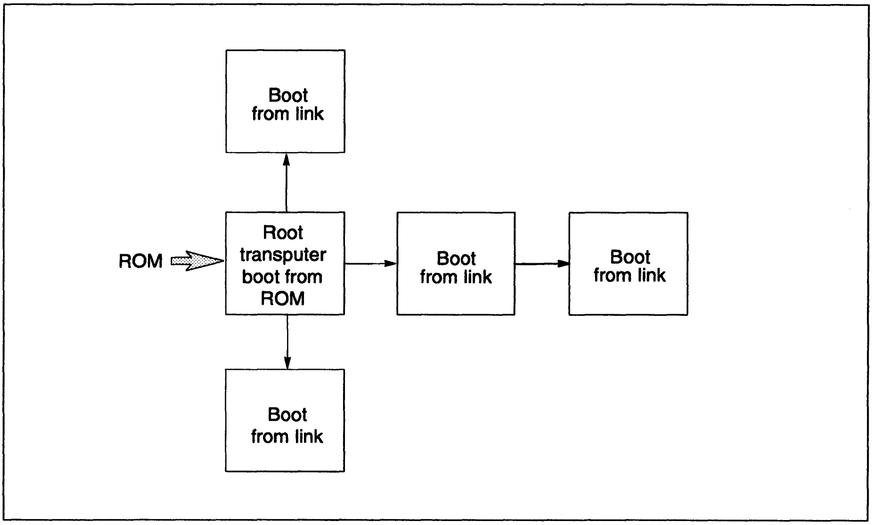


Figure 13.1 Loading a network from ROM

To prepare software to be booted from ROM, rather than to be booted from link, the following steps must be taken:

1 Rerun the configurer and collector tools with different options 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 13.2 and 13.3 illustrate the stages of preparing ROM-bootable software. Figure 13.2 shows a non-configured program, compiled and linked for a single processor.

Figure 13.3 shows a configured program, consisting of one or more linked units, connected by channels and allocated to processors as described in a configuration description file.

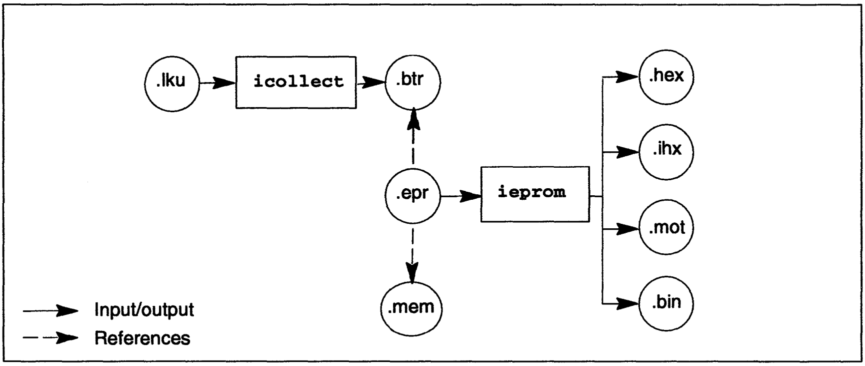


Figure 13.2 Preparation of ROM-bootable software (non-configured program)

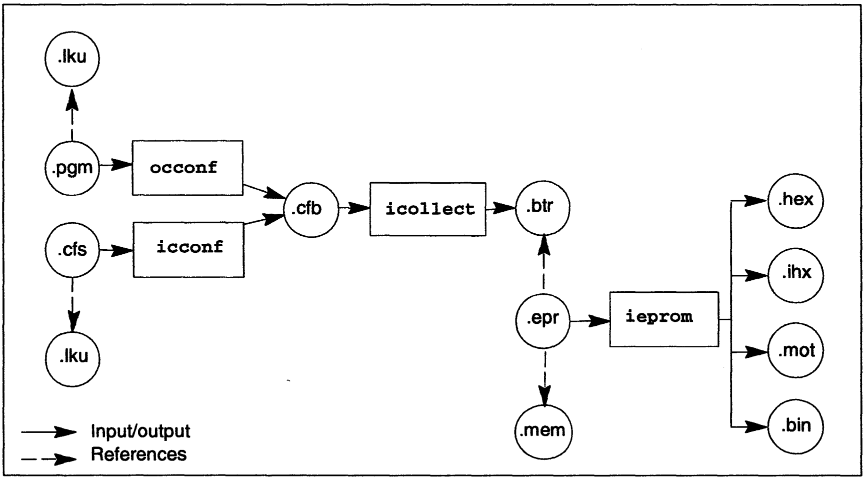


Figure 13.3 Preparation of ROM-bootable software (configured program)

`ieprom` is driven by a control file which normally has the file extension `.epr`. The control file describes the layout of the EPROM and gives the file names of the ROM bootable code file and any memory configuration file. A detailed description of `ieprom` and its control file, including examples, is given in chapter 7 of the accompanying *Toolset Reference Manual.*

13.2 Processing options

The processing options 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 options.

When preparing C programs 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 either be configured or prepared as a single, linked program.

13.2.1 Single processor, run from ROM

The application is defined as a collection of processes, connected as described in a configuration description 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.

13.2.2 Single processor, run from RAM

The application is defined as a collection of processes, connected as described in a configuration description 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.

13.2.3 Multi-processor, run from RAM

The application is defined as a collection of processes, connected and allocated to processors as described in a configuration description file. The compiled and configured application code is placed in the ROM of the root processor. When booted from ROM, the root processor copies 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 13.1).

13.2.4 Multi-processor, root run from ROM, rest of network run from RAM

The application is defined as a collection of processes, connected and allocated to processors as described in a configuration description 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. Each processor then sets off its own processes, and the application runs.

13.3 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 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**.

There are two forms of memory configuration file for two different types of memory interface supplied on different transputer versions. Memory configuration files for ST20450 transputers are created and edited using **imem450**. Memory configuration files for IMS T400, T414, T425, T800 and T805 transputers are created and edited using **iemit**.

The format of these files, and the editing tools **imem450** and **iemit** are described in the accompanying *Toolset Reference Manual.*

13.4 Producing 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 the configuration description 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 name of the root processor.

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 size, in bytes.

The collector will add the appropriate ROM bootstrap to the application code and the output file will be given the extension `.btr`.

13.5 Summary of EPROM tool steps for different configurations

13.5.1 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 `ieprom` to add memory interface (if needed), 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 `ieprom` to add memory interface (if needed), and produce EPROM files.
Multi-processor, run from RAM.	Compile and link a set of units, one per process.	Configure with the `RA`, `RS` and `P` options.	Collect	Run `ieprom` to add memory interface (if needed), and produce EPROM files.
Multi-processor, root runs from ROM, others from RAM.	Compile and link a set of units, one per process.	Configure with the `RO`, `RS` and `P` options.	Collect	Run `ieprom` to add memory interface (if needed), and produce EPROM files.

13.5.2 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 `ieprom` to add memory interface (if needed), and produce EPROM files.
Single processor, run from RAM.	Compile and link a set of units.	Configure with the `RA` option.	Collect	Run `ieprom` to add memory interface (if needed), and produce EPROM files.
Multi-processor, run from RAM.	Compile and link a set of units.	Configure with the `RA` option.	Collect	Run `ieprom` to add memory interface (if needed), and produce EPROM files.
Multi-processor, root runs from ROM, others from RAM.	Compile and link a set of units.	Configure with the `RO` option.	Collect	Run `ieprom` to add memory interface (if needed), and produce EPROM files.

14 ST20450 memory interface configuration

This chapter describes the process of designing a configuration for the ST20450 external memory interface, using the toolset software. It assumes that a memory system has already been designed, although it does consider some implications for memory design. Some illustrative examples are shown, which are not full worked examples, but show how the interface configuration relates to the requirements of the memory system.

The steps required to build and use the memory configuration code are described in section 14.9.

The ST20450 is also known as the T450, and is referred to by that names in the memory configuration.

The memory interfaces for T2/T4/T8-series transputers other than the ST20450 cannot be initialized by software, and so are not described in this chapter. The **`iemit`** tool is provided to assist hardware and ROM designers of boards using these processors. For details see the **`iemit`** chapter of the *Toolset Reference Manual*.

The design of configurations can be aided by the use of the memory interface configurer tool, **`imem450`**. This tool assists with selecting the parameters for an ST20450 external memory interface. The tool produces a memory interface configuration file, called a *memfile*, which describes the external memory configuration for a single processor. The memfile is used to create the initialization data as part of a host file or in a ROM. This is used to initialize the memory interface of the target hardware before code is loaded. Details of how to use the memfile to initialize the memory interface are given in chapter 4.

A memfile is normally needed to build the complete code for an application. It may be omitted if there is no external memory or if the information is already included in a ROM. Target hardware supplied with a boot ROM will normally have the memory interface initialized by the boot ROM, so the user will not need to be concerned with memory configuration. Target hardware supplied as a boot-from-link board will normally be supplied with an appropriate memfile.

The memory interface tool can create and modify memfiles interactively, displaying the effects that the parameters would produce. Alternatively, a memfile may be created or modified by an ordinary text editor. The **`imem450`** tool can also produce timing data or waveform diagram files for printing.

The **`imem450`** chapter of the *Toolset Reference Manual* describes how to use the memory interface tool and outlines its capabilities. Example displays are provided. The format of the memfile is given in an appendix to the *Toolset Reference Manual*.

Details of the ST20450 memory interface and the configuration registers may be found in the *ST20450 Datasheet*.

14.1 The memory interface

The ST20 has an *external memory interface* (EMI) which provides address decoding, timing control and refresh functions. It allows up to four banks of memory or other devices to be used.

The memory interface provides four *configurable banks*. Each configurable bank can be initialized to have a different interface with a different data bus width, timing and other characteristics. Each bank can have a data bus 32 bits, 16 bits or 8 bits wide. This width is known as the *port width*. Address decoding between the banks is performed internally by the memory interface, so in many cases no 'glue logic' is required. Each bank has its own timing signal strobes, so for each access only the appropriate bank is activated.

The address space is the same as the range of a 32-bit integer, with the most negative integer (#80000000) at the bottom of the address space and the most positive integer (#7FFFFFFF) at the top of the address space.

The ST20450 memory map is shown in figure 14.1. The four banks are at fixed addresses, each occupying a quarter of the total address space. Bank 2 contains the peripheral registers and is usually used for I/O devices. When the processor boots from ROM it starts executing code from address #7FFFFFFE, so Bank 3 is normally reserved for ROM.

The 16 Kbyte of on-chip memory acts as fast internal memory. The address range of the internal memory is fixed at the bottom of the address space, i.e. from #80000000 to #80003FFF. Any access to addresses in this range will access the internal memory, and external memory will not be accessed. The internal memory can be disabled if necessary.

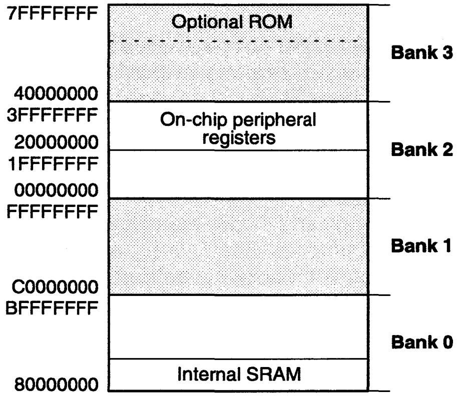


Figure 14.1 ST20 memory map

The peripheral registers, including the EMI configuration registers, are on-chip in the address space #20000000 to #3FFFFFFF. This space is reserved for such registers and cannot be accessed through the EMI. Any memory at these addresses cannot be used.

Internal subsystems request memory accesses from time to time, and the internal memory subsystem decides whether an external access is needed. It can request data from external memory or memory-mapped devices, or it can send data to external memory or memory-mapped devices. Such requests are handled by the memory interface.

Before the external memory can be used, the memory interface needs to know some details of the external memory, such as its timing and refresh requirements. This data is known as the memory interface configuration. This configuration data is held in configuration registers, which are written to during initialization.

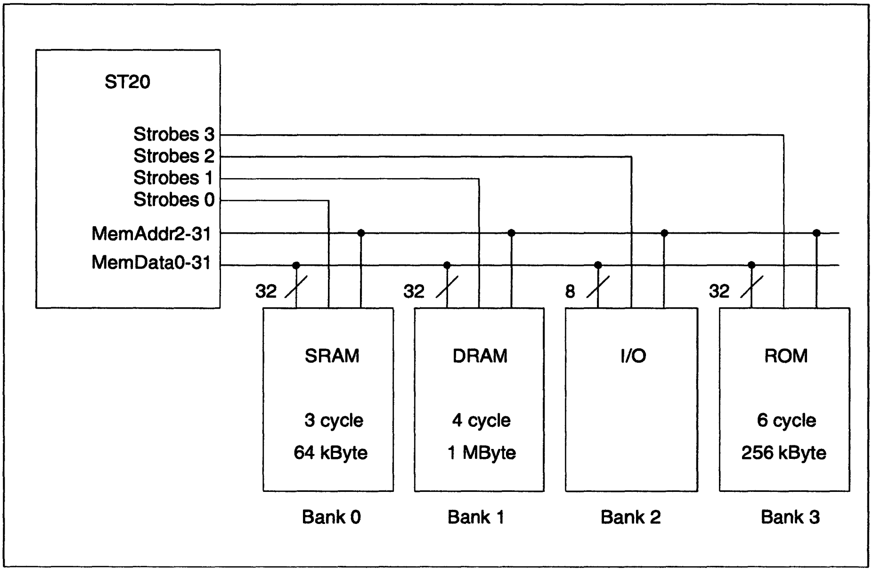


Figure 14.2 Example of a mixed memory system

The processor uses 32-bit addresses to address bytes, which gives an address space of 4 Gbytes. The memory interface has a 32-bit internal address bus. The address bus appears externally as 30 address pins and four byte-enable strobes, which are used to select one or more bytes within a word. For 8-bit and 16-bit memories, the unused byte strobe pins are used for the extra address pins.

For each of the four configurable banks, three separate programmable strobes are provided. They are given the names **notMemRAS**, **notMemCAS** and **notMemPS**. As the names suggest, when using DRAM these three strobes are normally used for RAS, CAS and, for example, Output Enable. They are completely programmable and their use can be varied depending on the requirements of the system being designed.

There are four byte-enable strobes (**notMemBE0-3**), all four of which are used by all the banks. These strobes are used to select which bytes will be read or written and are normally used for Write Enable. They are programmable in each bank, so the timing depends on the bank being addressed.

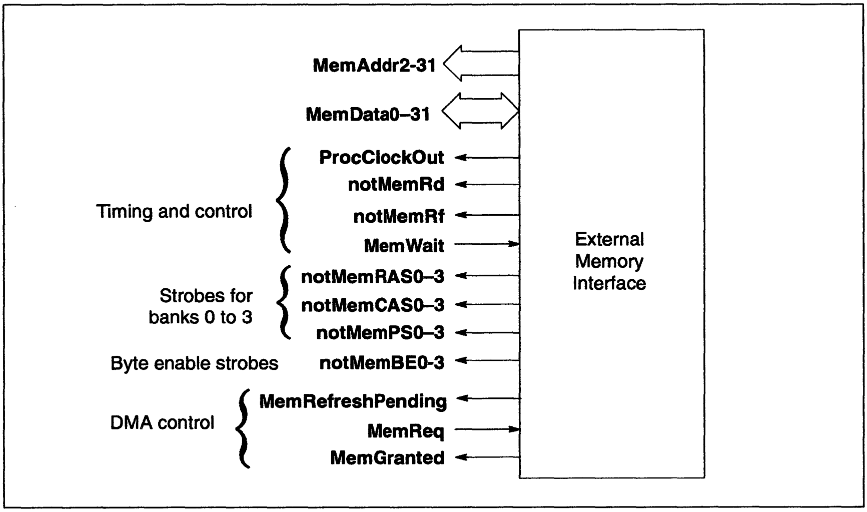
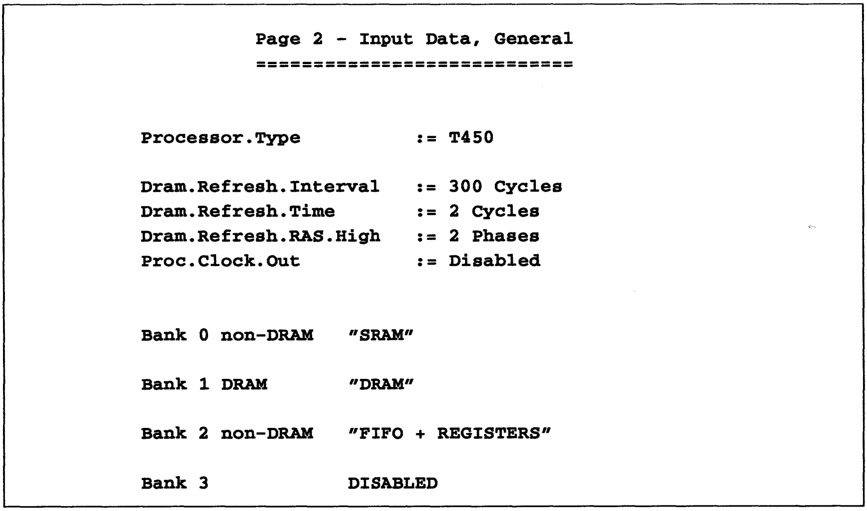


Figure 14.3 The memory interface pins



```
                  Page 2 - Input Data, General
                  ============================


        Processor.Type            := T450

        Dram.Refresh.Interval     := 300 Cycles
        Dram.Refresh.Time         := 2 Cycles
        Dram.Refresh.RAS.High     := 2 Phases
        Proc.Clock.Out            := Disabled


        Bank 0 non-DRAM    "SRAM"

        Bank 1 DRAM        "DRAM"

        Bank 2 non-DRAM    "FIFO + REGISTERS"

        Bank 3             DISABLED
```

Figure 14.4 Example `imem450` display page 2

In order to allow extension of memory cycles, a **MemWait** pin is provided that can add extra clock cycles to the current memory access when it is held high.

The **MemReq**, **MemGranted** and **MemRefreshPending** pins can be used to allow external devices to take over the address and data bus from the memory interface in order to perform DMA data transfers, for example.

14.2 General parameters

Figure 14.4 shows an example of page 2 of the `imem450` display, which includes general parameters which do not refer to any particular bank. The refresh parameters are discussed in section 14.6.2. Page 3 shows the pad strengths and pages 4 to 7 show the parameters for banks 0 to 3 respectively.

The four banks can each be marked as `DRAM` or `non-DRAM`, which determines possible parameters. The banks can also be given names, which appear on the timing data pages and the waveform diagrams.

14.2.1 Waveform diagrams

The `imem450` waveform diagrams should be used to check the general shape of the timing information.

In some cases it is important to check sequences of memory accesses, such as a sequence of reads from the same row of DRAM. Page 19 of the `imem450` display can show such a sequence. The sequence of accesses is controlled by the input data shown on page 8, which defines the bank, type and address for each of up to eight accesses.

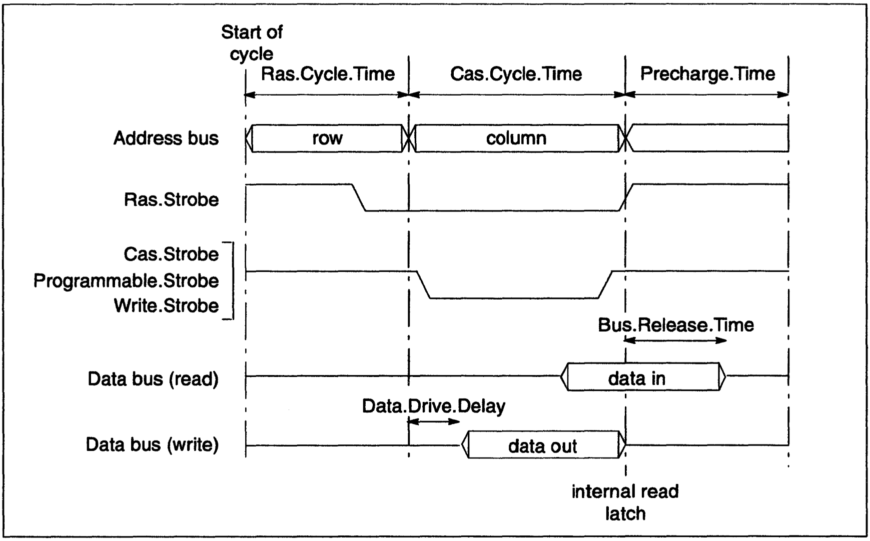


Figure 14.5 DRAM memory cycle

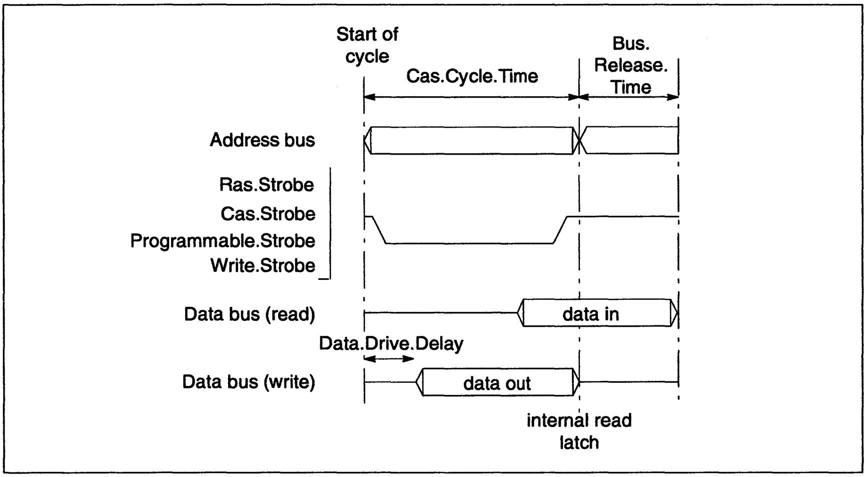


Figure 14.6 Non-DRAM memory cycle

14.3 Timing

The full memory cycle is divided into three sub-cycles, `Ras.Cycle.Time`, `Cas.Cycle.Time` and `Precharge.Time`, as shown in figure 14.5. The names `Ras.Cycle.Time`, `Cas.Cycle.Time` and `Precharge.Time` suggest the usual use of the three sub-cycles with DRAM, but they have no special significance.

When using SRAMs and other memories and devices, it is rarely necessary to define a `Ras.Cycle.Time` for the memory cycle; the `Cas.Cycle.Time` becomes the whole of the memory cycle, as shown in figure 14.6.

All timing parameters are given in either processor clock *cycles* or *phases*. For a processor of speed m MHz, one cycle is $1/m$ microseconds. A phase is half a cycle. If the processor speed is m MHz then:

$$\begin{aligned} n\ cycles &= \frac{n}{m}\ microseconds \\ p\ phases &= \frac{p}{2m}\ microseconds \\ x\ nanoseconds &= x \times m \times 10^{-3}\ cycles \\ &= 2\,x \times m \times 10^{-3}\ phases \end{aligned}$$

Timings for sub-cycles such as `Ras.Cycle.Time`, `Cas.Cycle.Time`, `Precharge.Time` and `Bus.Release.Time` are given in cycles, whereas edge related timings, namely `Time.to.Falling.Edge`, `Time.to.Rising.Edge` and `Ras.Edge.Time`, are defined in phases.

14.3.1 Strobes

A set of timing strobes is provided for each bank for use as timing signals. They can be configured to become active at appropriate times. The strobes are provided by the pins

notMemCAS0-3, **notMemRAS0-3** and **notMemPS0-3**. In addition there is a set of four byte enable strobes, **notMemBE0-3**, used to select the bytes being accessed, which all apply to all four banks. The names of the strobes used by `imem450` correspond to these pins, as shown in table 14.1.

Pin name	Strobe
notMemCAS0-3	`Cas.Strobe`
notMemRAS0-3	`Ras.Strobe`
notMemPS0-3	`Programmable.Strobe`
notMemBE-3	`Write.Strobe`

Table 14.1 Strobe names

Appropriate names can be given to the strobes. These names have no significance except that they appear on the timing waveform diagrams.

Suitable timing strobes should be selected to drive the control pins of the devices in the bank. The strobes `Cas.Strobe`, `Programmable.Strobe` and `Write.Strobe` are similar in that they start high and can be made to fall and rise during the `Cas.Cycle.Time`. If the rising edge parameter implies that the rising edge will be after the end of the CAS cycle time, then the strobe will rise at the end of the CAS cycle time. If the rising and falling edges coincide or the rising edge is before the falling edge or the falling edge is after the end of the cycle time, then the strobe will be inactive.

The `Ras.Strobe` is similar but differs in that it can be made to fall during `RAS.Cycle.Time`, as described in section 14.6.1. This facility is provided so that it can be used to latch the RAS address on DRAMs.

The `Write.Strobes` are usually used to drive the read/write pins. The timing of the `Write.Strobes` is defined separately for each bank, although it refers to the same set of write strobe pins common to all the banks. The timing of these strobes depends on which bank is being accessed.

14.3.2 Timing skews

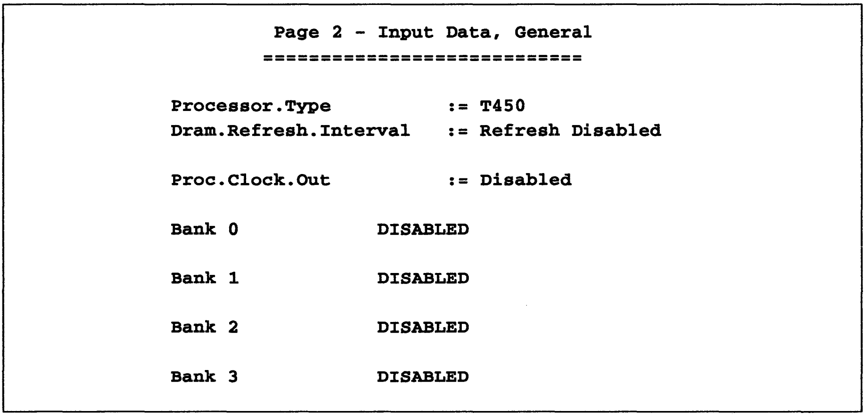
Each rising or falling timing strobe edge is specified relative to either the start of `Ras.Cycle.Time` or the start of `Cas.Cycle.Time`. Each of these edges is subject to timing skew depending on the external loading of the memory interface pins. These skews must be considered when designing the interface configuration.

As well as skews due to the memory interface, other sources of skew and delay should also be taken into account when calculating timing parameters. For example, if a bank is sub-decoded using logic gates, then the propagation delay through the gates will delay signal edges.

14.4 Configuring for no external memory

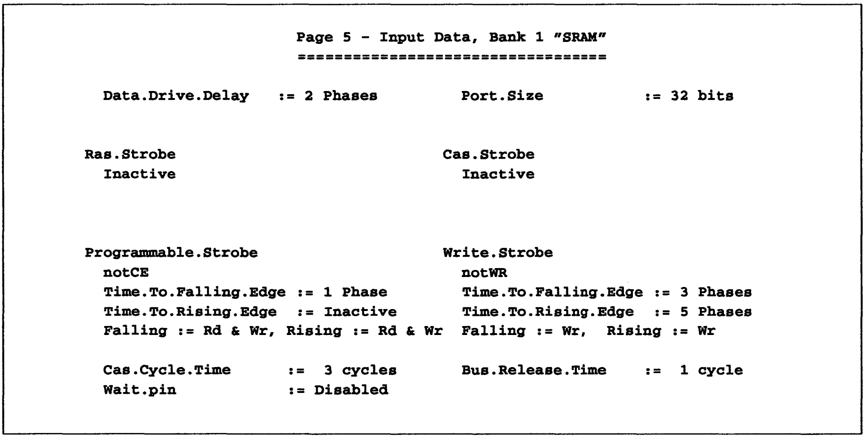
Many ST20 family devices can be used without any external memory or external I/O, since a communication link can be used to set up and bootstrap the device, and internal memory is provided on-chip.

If there is no external memory then no memfile is needed, but a memfile may be supplied, in which case the memory should be configured as in figure 14.7.



```
                    Page 2 - Input Data, General
                    ============================

          Processor.Type           := T450
          Dram.Refresh.Interval    := Refresh Disabled

          Proc.Clock.Out           := Disabled

          Bank 0            DISABLED

          Bank 1            DISABLED

          Bank 2            DISABLED

          Bank 3            DISABLED
```

Figure 14.7 Example `imem450` display for no external memory



```
                              Page 5 - Input Data, Bank 1 "SRAM"
                              ==================================

     Data.Drive.Delay   := 2 Phases          Port.Size          := 32 bits


   Ras.Strobe                              Cas.Strobe
     Inactive                                Inactive



   Programmable.Strobe                     Write.Strobe
     notCE                                   notWR
     Time.To.Falling.Edge := 1 Phase         Time.To.Falling.Edge := 3 Phases
     Time.To.Rising.Edge  := Inactive        Time.To.Rising.Edge  := 5 Phases
     Falling := Rd & Wr, Rising := Rd & Wr   Falling := Wr,  Rising := Wr

     Cas.Cycle.Time      :=  3 cycles        Bus.Release.Time    :=  1 cycle
     Wait.pin            := Disabled
```

Figure 14.8 Example display page for SRAM

14.5 Configuring for SRAM

SRAM (Static RAM) banks are normally marked as `non-DRAM`. The software assumes that non-DRAMs will have the following characteristics:

- No refresh is needed.

- Address pins are not multiplexed.

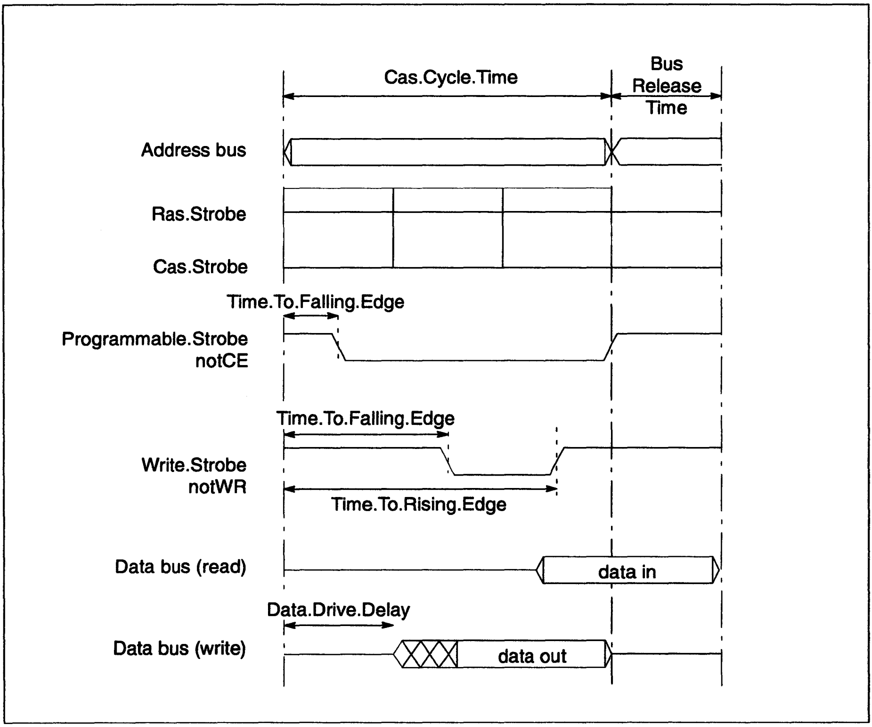


Figure 14.9 Example SRAM timing

- Paged memory is not supported.
- No precharge time is needed.

Figure 14.8 shows an example of the `imem450` input for 64 Kbytes of SRAM in bank 1. The SRAM is placed at the bottom of the address space, so the base address is #80000000.

RAS strobe and CAS strobe are not needed, but the Programmable Strobe is used to drive **notCE**.

14.5.1 Timings

Figure 14.9 shows the timing roughly corresponding to the SRAM parameters in figure 14.8, showing the meanings of the timing parameters.

The bus release time is the minimum time after a read access to allow the memory to release the data bus. It is the minimum time after a read access before the start of:

- a write access or
- a read access from another bank.

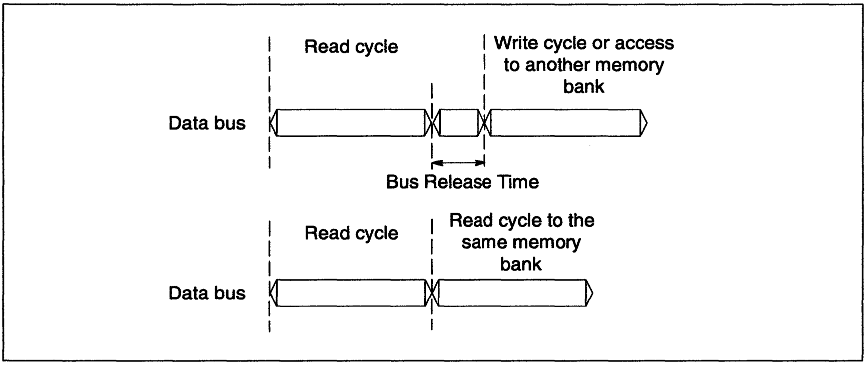
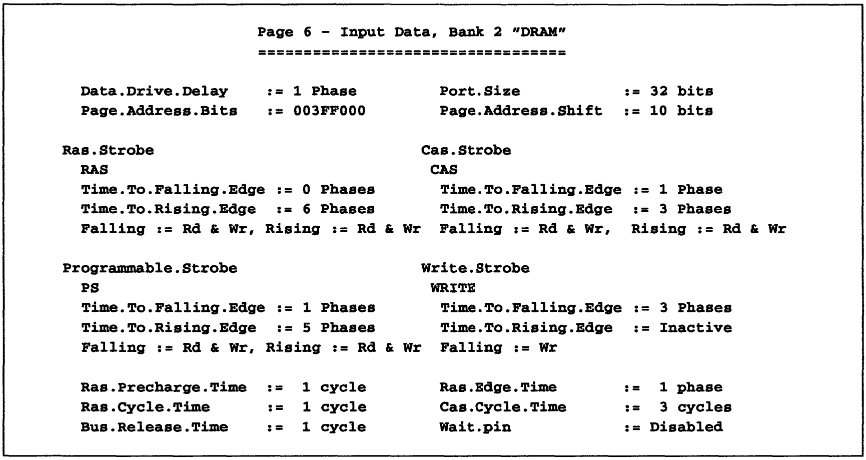

Read cycle
Write cycle or access to another memory bank
Data bus
Bus Release Time
Read cycle
Read cycle to the same memory bank
Data bus

Figure 14.10 Addition of **BusReleaseTime** between memory cycles

The data drive delay is the time after the start of the write cycle before the data is put on the bus. These two delays ensure that the memory interface will not attempt to use the data bus until the device has completely switched off its output drivers.

The wait pin, **MemWait,** may be used to dynamically delay access times. This may be used when devices of different speeds are used in the same bank. In this case, the parameters used would generally be the parameters for the fastest devices, and the wait pin would be used to slow down the cycle for the slower devices. If the **MemWait** pin is not used for this bank, then the bank `Wait.Pin` parameter should be marked as disabled.



```
                     Page 6 - Input Data, Bank 2 "DRAM"
                     ==================================

   Data.Drive.Delay     := 1 Phase          Port.Size            := 32 bits
   Page.Address.Bits    := 003FF000         Page.Address.Shift   := 10 bits

 Ras.Strobe                               Cas.Strobe
   RAS                                     CAS
   Time.To.Falling.Edge := 0 Phases         Time.To.Falling.Edge := 1 Phase
   Time.To.Rising.Edge  := 6 Phases         Time.To.Rising.Edge  := 3 Phases
   Falling := Rd & Wr, Rising := Rd & Wr    Falling := Rd & Wr,  Rising := Rd & Wr

 Programmable.Strobe                      Write.Strobe
   PS                                      WRITE
   Time.To.Falling.Edge := 1 Phases         Time.To.Falling.Edge := 3 Phases
   Time.To.Rising.Edge  := 5 Phases         Time.To.Rising.Edge  := Inactive
   Falling := Rd & Wr, Rising := Rd & Wr    Falling := Wr

   Ras.Precharge.Time   :=  1 cycle         Ras.Edge.Time        :=  1 phase
   Ras.Cycle.Time       :=  1 cycle         Cas.Cycle.Time       :=  3 cycles
   Bus.Release.Time     :=  1 cycle         Wait.pin             := Disabled
```

Figure 14.11 Example `imem450` display page for DRAM

14.6 Configuring for DRAM and Video RAM

Any bank may be marked as `DRAM` (Dynamic RAM), which means that refresh, address multiplexing and page mode are normally provided, though all of these can be switched off. Video RAMs are normally configured in the same way as DRAM.

Figure 14.11 shows an example of the `imem450` input page for 8 Mbytes of DRAM in bank 2.

In this example, the memory consists of 8 DRAM parts, each arranged as 1M x 4. Each has 10 multiplexed address pins. The memory is 32 bits (or 4 bytes) wide, so the two least significant bits of the 32-bit address select a byte within the width of the memory. The column address is the ten bits 2 to 11 and the row address is the ten bits 12 to 21, as shown in figure 14.12.

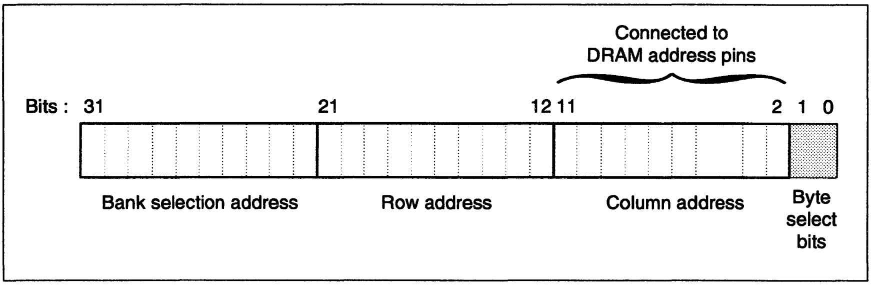


Figure 14.12 Example DRAM addressing

Three parameters, `Ras.Cycle.Time`, `Page.Address.Shift` and `Ras.Edge.Time` are provided to support multiplexed addresses. The parameter `Page.Address.Bits` is provided to support page mode. Multiplexed addresses are disabled if `Ras.Cycle.Time` is zero and page mode is disabled by adding `Disable.Page.Mode`.

The `Page.Address.Shift` is the number of bits the row address must be shifted to place it on the DRAM address pins. This is generally equal to the number of address pins, which is 10 in this example.

Two addresses in this bank are in the same page (i.e. the same row) if bits 12 to 21 are the same. The `Page.Address.Bits` parameter is the mask for the page address bits, i.e. the mask with bits 12 to 21 set, which is #003FF000 in this example.

As with SRAM, the wait pin, **MemWait**, may be used to dynamically delay access times. This may be used when devices of different speeds are used in the same bank. If the **MemWait** pin is not used for this bank, then the bank `Wait.Pin` should be marked as disabled.

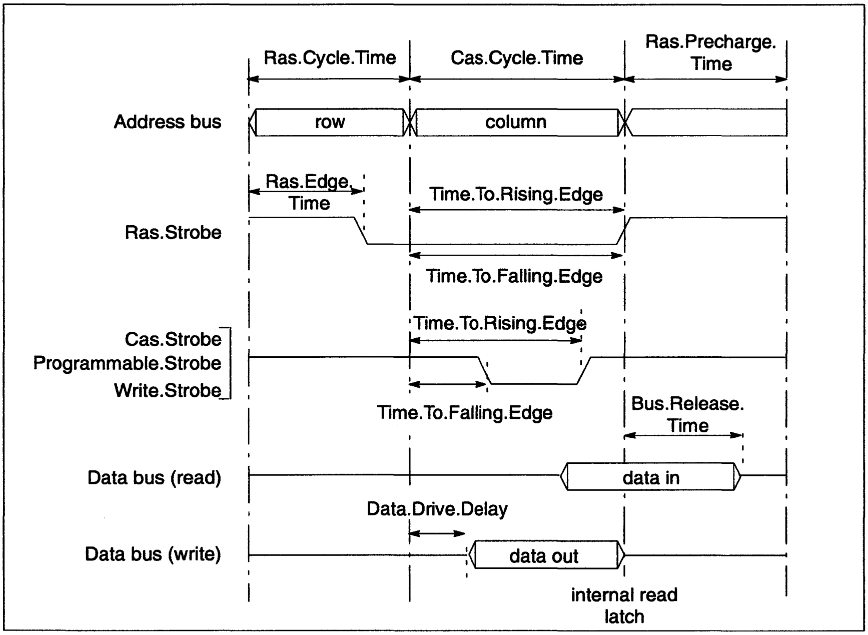


Figure 14.13 Example DRAM timing

14.6.1 Timings

Figure 14.13 shows the timing for DRAM, showing the meanings of the timing parameters. DRAMs with non-multiplexed address pins should have `Ras.Cycle.Time` set to zero.

`Ras.Strobe` is different from the other strobes because it can be made to fall within `Ras.Cycle.Time`. This is done by setting `Ras.Edge.Time`, as shown in figure 14.13. `Ras.Strobe` will then be low at the start of `Cas.Cycle.Time`. By default, `Ras.Strobe` will rise again at the end of `Cas.Cycle.Time`. To achieve this, `Time.To.Rising.Edge` and `Time.To.Falling.Edge` should both be set to `To end of cycle` or the `Time.To.Falling.Edge` can be set to zero.

If an earlier rise is needed then this can be set using `Time.To.Rising.Edge`, as shown in figure 14.14. In this case, `Time.To.Falling.Edge` should be set to less than `Time.To.Rising.Edge` or greater than the `Cas.Cycle.Time` to avoid the `RAS.Strobe` falling again.

The `Ras.Strobe` is also capable of rising and falling a second time in the same cycle, as shown in figure 14.14. This is not normally required and the times to rising and falling edges should be set to end of cycle to avoid this happening.

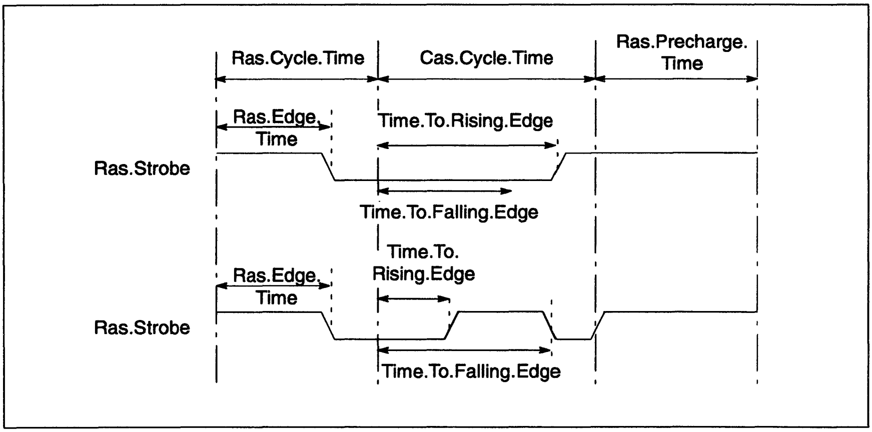


Figure 14.14 Alternative **Ras.Strobe** timings

Ras.Precharge.Time is the time when DRAM charges its internal capacitors. A precharge time will occur before another access to the same bank if:

- the next access is to a different row or
- the next access is to a different bank.

When a precharge sub-cycle occurs, all the strobes for that particular memory bank are reset inactive and are held in that state until the end of the sub-cycle. Since the DRAMs are not selected during precharge, the address and data buses can be used to access other memory banks, thus allowing the precharge to be performed at the same time as accesses to other banks.

The **Bus.Release.Time** and **Data.Drive.Delay** are used to allow time for the memory to release the data bus. It allows the designer to ensure that the ST20 will not attempt to use the bus until the device has completely switched off its output drivers. **Bus.Release.Time** starts at the end of the **Cas.Cycle.Time** and can occur at the same time as the **Ras.Precharge.Time**. **Data.Drive.Delay** starts at the beginning of the next **Cas.Cycle.Time** and delays the output of data if it is a write cycle. **Bus.Release.Time** plus **Ras.Cycle.Time** plus **Data.Drive.Delay** is the minimum time after a read access before the data bus can be driven in:

- a write access or
- a read access from another bank.

14.6.2 Refresh

The refresh control parameters on page 2 of the **imem450** display control the refresh timings associated with driving DRAMs. There is one set of parameters which controls refresh for all the DRAM banks. Banks marked as `non-DRAM` or with the `Disable.Refresh` flag are not refreshed. If two or more types of DRAM are present with different refresh requirements then the parameters must be set to values which satisfy all the DRAMs.

Page 2 contains three refresh timing parameters, namely `Dram.Refresh.Interval`, `Dram.Refresh.Time` and `Dram.Refresh.Ras.High`, and one flag, `Signal.All.Pending.Cycles`, as shown in figure 14.15. The timing of a refresh cycle is shown in figure 14.16.

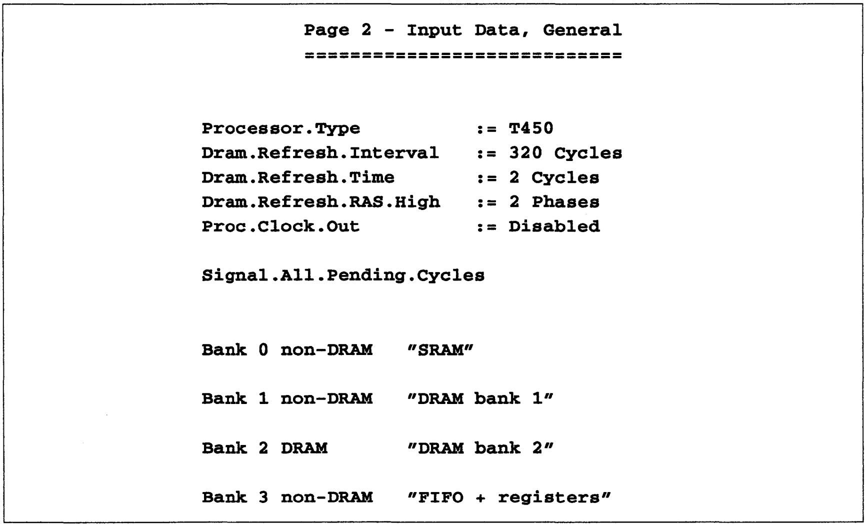
`Dram.Refresh.Interval` is the time, in cycles, between successive refresh cycles. One row of each bank is refreshed in each refresh cycle, so if the time between refreshes of any one row in milliseconds is *Time.between.refreshes* then:

`Dram.Refresh.Interval`
= *Time.between.refreshes* x *m* x 1000 ÷ *number.of.rows*

where *m* is the processor speed in MHz.

`Dram.Refresh.Time` is the time from the falling edge of CAS before RAS and CAS can be taken high during a refresh cycle. This value is given in cycles. If it is set to zero, then no refreshing will occur.

`Dram.Refresh.Ras.High` is the time of the falling edge of the RAS strobe during a refresh cycle after the CAS strobe goes low. This value is given in phases.



```
                        Page 2 - Input Data, General
                        ============================


              Processor.Type             := T450
              Dram.Refresh.Interval      := 320 Cycles
              Dram.Refresh.Time          := 2 Cycles
              Dram.Refresh.RAS.High      := 2 Phases
              Proc.Clock.Out             := Disabled

              Signal.All.Pending.Cycles


              Bank 0 non-DRAM   "SRAM"

              Bank 1 non-DRAM   "DRAM bank 1"

              Bank 2 DRAM       "DRAM bank 2"

              Bank 3 non-DRAM   "FIFO + registers"
```

Figure 14.15 Example `imem` display showing refresh parameters

`Signal.All.Pending.Cycles` defines the meaning of the **MemReqOut** pin signal. This is generally used when an external device has direct access to the memory (DMA).

With the `Signal.All.Pending.Cycles` flag set, the **MemReqOut** pin is used to signal that a memory access or refresh is waiting. Otherwise, **MemReqOut** is used to signal that a refresh is due.

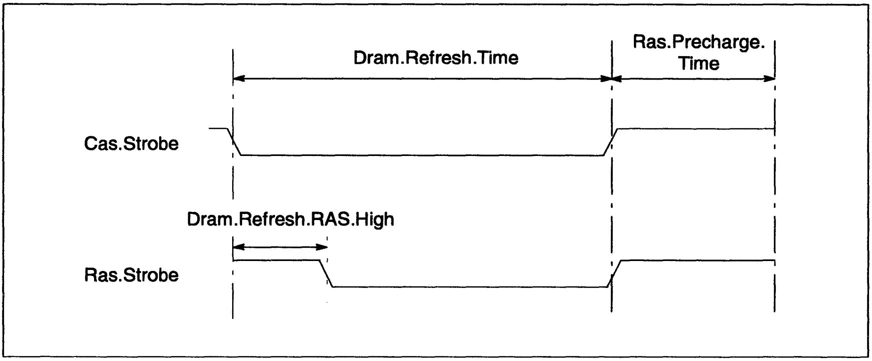


Figure 14.16 Refresh timings

14.7 Configuring for ROM

14.7.1 EPROM

EPROMs are devices that contain a programmable non-volatile memory array. These can be easily programmed to bootstrap a single processor or a multi-processor system, using the relevant software tools to generate the programming data. EPROMs are often byte-wide, but may be wider.

EPROM is configured as non-DRAM, in much the same way as SRAM. The strobes should all be marked as active on read cycles only. By convention, the `Programmable.Strobe` is used to drive the notCE pin. The write strobes are not needed.

14.7.2 Flash EPROM

This type of ROM is electrically erasable, but is non-volatile in the same way as EPROM. Flash EPROM allows the system to change its own bootstrap code, for example in a prototype system.

Flash EPROM is configured as non-DRAM, in much the same way as SRAM. The strobes can all be marked as active on read and write cycles except the write strobes.

14.8 Configuring for non-memory devices

Non-memory devices are generally configured as non-DRAM.

The code being used to access the device should be carefully matched to the width of the port. For example, reading or writing a word will cause four accesses to a byte-wide bank. The order of such accesses is not guaranteed.

It may be desirable to configure the bank to wider than the device in order to use block moves to read or write. The order of accesses within a block move is not guaranteed. However, the minimum possible number of accesses will be made, even if an interrupt occurs.

Otherwise, configuring for I/O devices is similar to configuring for SRAM.

14.9 Building and using memory configuration code

ST20450 memory interface configuration code is generated from a memfile and must be run on the processor before any external memory can be used. The memory interface configuration code can be in different forms depending upon the method of booting the target hardware. The different methods of booting each ST20450 are:

- booting the ST20450 from link with no ROM present;
- booting the ST20450 from a ROM which contains application code;
- booting the ST20450 from a memory configuration ROM and then simulating a boot-from-link.

All SGS-THOMSON T450 TRAMs include a memory configuration ROM. Such TRAMs may be used as if they were booting from link with the memory interface already configured.

The method of building and using the memory configuration code in each of these cases is described in the following sections.

The memory configuration code is small enough to run in the internal memory of the ST20. It initializes the memory interface and then terminates. The application can then be loaded, provided that the processor is not reset.

For multi-processor applications, a memory configuration ROM may be used for each non-root ST20450. Alternatively the example scripts or batch files described in section 14.9.1 may be modified to build a multi-processor memory bootable or ROM code.

14.9.1 Booting from link

A separate bootable file, called a *memory bootable*, is loaded and run, using `irun`, before the application bootable is loaded. Either the `-sc` option can be used on the `irun` command line or the path name of the memory bootable may be added to the extra parameters field of the AServer database, as described below.

The memory bootable configures the memory interface and then simulates the boot-from-link state so that the application bootable code can be loaded. The memory bootable will run in internal memory, so it does not need access to external memory.

A memory bootable may be generated using the example script `buildma` for Sun users, or the example batch file `buildma.bat` for PC users. These examples are provided to help with the generation of such bootables and may be found in the `example` directory of the release. For example, the following command will build a memory bootable called `mymemcfg.btl` from the memfile `mymemcfg.mem`:

```
buildma mymemcfg
```

Running a memory bootable

The memory bootable and the application bootable can be loaded with a single `irun` command. `irun` will automatically load a memory bootable before loading the application provided the memory bootable is given in the AServer database file. AServer database files are described in section 8.5. The AServer database file should have been set up as part of the installation, and the memory bootable will not normally change unless the hardware is changed.

For example, if the memory bootable is `mymem.btl` then `mymem.btl` must be added in the extra parameters field for the line for the appropriate target hardware connection. A complete AServer database line for a connection named `ST20` might be:

```
| ST20 | txcs b008p.DLL "#150 #B" | 1 | mymem.btl
```

Alternatively, the memory bootable file may be explicitly loaded with the `irun` option SC. For example, to load the memory bootable `mymem.btl` and then load and run the application `app.btl`:

```
irun -sc mymem.btl app.btl
```

If memory bootables are given both in the AServer database and on the `irun` command line then the one given in the AServer database will be run and then the one given on the command line. The memory bootable on the command line will therefore overwrite the effect of the one given in the AServer database.

14.9.2 Booting from an application code ROM

The ROM which contains the application code also includes the memory interface configuration code. The ROM is generated using the EPROM tool `ieprom`, as described in chapter 13. The memfile name must be given in the `ieprom` control file, using the `memory.configuration` statement. For example, to include code derived from the memfile `mymemcfg.mem`, the following line should be included in the control file:

```
memory.configuration mymemcfg.mem
```

14.9.3 Booting from a memory configuration ROM

A ROM is used which contains only the memory interface configuration code. The code in the ROM configures the memory interface and then simulates booting from link so that the application bootable may be loaded from a link.

The ROM code may be generated using the example script `buildmr` for Sun users, or the example batch file `buildmr.bat` for PC users. These examples are provided to help with the generation of such ROM code and may be found in the `example` directory. For example, the following command will build ROM code from the memfile `mymemcfg.mem`:

```
   buildmr mymemcfg
```

This produces the file `mymemcfg.hex` which is suitable for blowing into ROM.

15 Performance improvement

15.1 Introduction

This chapter describes ways in which to improve the performance of occam programs which have been built with the occam 2.1 toolset.

Many of the techniques described here are equally applicable to early occam toolsets, and indeed to many other languages and computer systems. Similarly, many of the transputer-specific optimizations are relevant when programming transputers in other languages.

Also discussed are the specific features of the transputer which are amenable to optimizations, and how to use the toolsets to take advantage of them. Transputers covered include the IMS T212, T222, T225, T400, T414, T425, T800, T801, T805 and ST20450.

15.1.1 Transputer architecture

This chapter will not attempt to describe the transputer architecture. However, the particular points to note about the transputer, when considering performance improvement, are as follows:

- On-chip RAM.

  Each transputer has a part of its address space implemented as on-chip RAM, which means that it can be accessed very quickly. There is a noticeable penalty in accessing external RAM. Much of this chapter describes methods to ensure that best use is made of this on-chip RAM.

- Instruction prefixing.

  Transputers use a variable length instruction encoding, which is built up out of lots of single byte instructions. It is useful to minimize the size of these instructions, both to minimize the code space required, and to minimize the time taken to fetch the instructions from memory.

In practice, the only instructions whose length can be easily controlled are those which access local variables; hence it is the layout of local variables which is important.

15.2 Trade-offs and issues

15.2.1 Space versus time

Most optimizations which are performed are intended to minimize the running time of a program. This is known as optimizing for time. In certain circumstances it is required to minimize the size of a program; either code size, data size, or both. This is known as optimizing for space. Often a particular optimization will produce an improvement in both space and time. In general, and in this chapter, most optimizations are aimed to optimize for time.

15.2.2 On–chip RAM

The on-chip RAM is based at the bottom of a transputer's memory address space. This implies that it is important to utilize this space properly. In some cases, the whole program can fit inside this RAM. In many other cases, however, a decision must be made as to the best use of this RAM.

As a general rule of thumb, the *stack* (or *workspace*) of a program should be placed onto the on-chip RAM if possible. This is because the transputer instruction encoding makes data accesses more frequent than instruction accesses, so less penalty will be incurred if the data resides in fast memory. If there is space, then it is useful to put inner loops and other frequently used code subroutines into on-chip RAM too.

The occam toolsets attempt to place workspace onto on-chip RAM by default.

Some TRAMs (Transputer Modules) are constructed in such a way that the lowest external memory in the address space is the next fastest, followed by slower RAM at higher addresses. Thus the bottom 4K might be on-chip RAM, the next 32K might be 3-cycle external SRAM, followed by 2M of 4-cycle external DRAM. The way these are treated is exactly analogous to the simpler case; by simply attempting to move the most important data and code areas to the bottom of memory.

15.2.3 Basic code generation techniques

The compiler supplied with the occam 2.1 toolset generates good code for expressions, but does not attempt to optimize code across statement boundaries; future compilers will. It may be useful to bear this in mind to try to improve performance; for example, by the introduction of abbreviations for commonly used expressions.

The occam compiler allocates memory statically; that is, given any program the compiler can determine exactly how much memory is required. This enables the loader to specify exactly how much workspace is required and to attempt to place it on-chip.

15.2.4 Processor classes and types

The occam 2.1 compiler can create code which can execute on many different types of transputer; these are known as transputer classes. This facility can be useful to build libraries which can be used for any transputer type. However, compiling for a particular transputer type will make most use of the instruction set of that transputer, and therefore will make a program execute faster.

It is worth noting that you can create a library which contains, for example, both TA and T425 code. The compiler and linker will automatically select the most specific modules which exist in that library, depending on the command line options supplied to the compiler or linker. Similarly if both interactive debugging and non-interactive debugging modules exist in a library, the most specific one (non-interactive) will be chosen.

The rule to use is: always compile and link for the specific transputer type to get the best performance.

15.2.5 Interactive debugging

The occam 2.1 toolset includes an interactive debugger for executing a program under user control, allowing breakpointing and other debugging operations. This involves a certain amount of overhead, both of space and time, because the compiler inserts run-time library calls for certain operations; these require extra code, and extra stack space. Once a program has been developed, this facility can be disabled to improve program performance.

Disabling interactive debugging has two effects. Firstly, it will improve the speed of code which performs communication, because the interactive debugger requires nearly all communication to be performed via library routines. The second effect is to reduce the memory size of the program; this reduces both workspace size (because the library routines would use stack space) and code size (because the library routines are not linked in). Also, when interactive debugging is enabled, a generic start-up routine must be inserted. Configuring a program with interactive debugging disabled, or collecting a single processor program with interactive debugging disabled, means that simpler start-up code can be used.

The rule to use is: always compile, link and configure with interactive debugging disabled to get the best performance.

15.2.6 Virtual routing

This occam configurer provides the ability to use *virtual routing*. This means that messages from one transputer to another can be automatically through-routed via intervening transputers without requiring to be explicitly programmed. Also, many occam channels may be multiplexed down a single transputer link.

This is performed by library routines which are inserted by the compiler. These are the same routines as are used by the interactive debugging system described in section 15.2.5. Therefore, to use the virtual routing facilities you must not disable interactive debugging.

When virtual routing is used, extra processes are placed onto each processor by the configurer as required. These will use up memory. Therefore if memory space is at a premium, it will be better not to use the virtual routing facilities. The configurer will automatically determine when the multiplexing and through-routing are not required, and will omit the processes. A command line switch NV on the configurer can be used to forcibly prevent the creation of these processes; in this case an error is issued if they would have been required.

15.2.7 Error modes

occam programs can be compiled in three different error modes; HALT, STOP, and UNIVERSAL. In HALT mode, as soon as any erroneous process is executed, the whole processor halts. This is implemented by making use of the transputer's global **HaltOnError** flag. In STOP mode, only that single process is halted. This requires the **HaltOnError** flag to be clear. UNIVERSAL mode is provided to act like either HALT or STOP, depending on the state of the **HaltOnError** flag.

Because of the behavior of particular transputer instructions, HALT error mode is the simplest and fastest to implement. STOP mode requires extra instructions to be inserted to detect and act on errors. UNIVERSAL mode requires slightly more instructions again.

UNIVERSAL mode does *not* switch off all run-time error checking. The `U` command line switch can be used to remove error checking code; see section 15.4.1. Thus any error mode, in combination with the `U` option, provides the occam UNDEFINED error mode.

The compiler libraries are most efficient in the HALT error mode, so benchmark programs, and programs *which are known to be correct*, should be compiled in HALT error mode, and with the `U` option.

The rule to use is: always compile, link and configure in the HALT error mode, and compile and configure with the `U` option, to get the best performance.

15.2.8 Vector space

The occam compiler uses a technique known as *separate vector space* to try to minimize a program's stack requirement. Arrays are placed into a separate area of memory known as the vector space. This is organized as another stack, in another area of memory. The idea is that the normal stack (workspace), containing local variables and the procedure call stack, will then be as small as possible, and will fit into on-chip RAM. In many cases the time required to access arrays will be less critical than the time to access local variables, so this provides a useful optimization.

The use of separate vectorspace requires that an extra (hidden) parameter is passed to each subroutine. In some circumstances this extra cost exceeds the benefit, so it might be useful to disable the separate vectorspace (using the compiler `V` option). Similarly, if the combined workspace and vectorspace would together fit into on-chip RAM anyway, it will be most efficient to disable the separate vectorspace.

It may be the case that access to most arrays is not critical, but that access to a particular array is extremely time critical. This single array can be retained in workspace (and hence more likely to be placed onto on-chip RAM) by a compiler *allocation*:

```
PLACE name IN WORKSPACE:
```

Alternatively, a program may consist of many small arrays which would benefit from being placed in workspace, plus a few large arrays which would not. In this case, separate vector space can be disabled by default. The large arrays can then be explicitly placed into vectorspace by another *allocation*:

```
PLACE name IN VECSPACE:
```

Finally, the configurer provides attributes which allow the whole vectorspace to be placed at the bottom of memory, etc; see section 15.4.5.

15.2.9 Alias checking

occam has strict rules about aliasing of variables, to ensure that different variables do not point to the same data. These rules enable the compiler to make better deductions about the behavior of a program, and therefore to generate better code. They also provide a simple model of what a section of program means; its behavior is not affected by the context in which it is executed.

The occam 2.1 toolset permits a programmer to disable alias checking, but at the programmer's own risk. In general, it will be better not to. There are two levels of control over alias checking:

- On a whole program at a time. Alias checking may be disabled for a whole program, by using the `A` command line switch, or `#OPTION`. The compiler will assume that all variables may alias each other.
- For individual variables. `#PRAGMA PERMITALIASES` may be used to indicate individual variables which may be aliased. All other variables are assumed to abide by the occam rules.

By default, the compiler will ensure that no aliases are permitted. In some cases, this can require code being generated to check at run-time. The WO option will cause a warning message to be generated whenever a run-time check is inserted. Where alias checks are disabled, these checks will not be generated. However, the compiler will have to make worse assumptions about the behavior of the program, and may generate slower code.

15.2.10 Usage checking

occam has strict rules about the usage of variables and channels in parallel processes. These rules enable the compiler to make better deductions about the behavior of a program, and therefore to generate better code. They also provide a simple model of what a section of program means; its behavior is not affected by the context in which it is executed.

The occam 2.1 toolset permits a programmer to disable usage checking, but at the programmers own risk. In general, it will be better not to. There are two levels of control over usage checking:

- On a whole program at a time. Usage checking may be disabled for a whole program, by using the `N` command line switch, or `#OPTION`. The compiler will assume that all variables may be accessed in parallel, synchronized by communications down channels.
- For individual variables. `#PRAGMA SHARED` may be used to indicate individual variables which are used in parallel processes. All other variables are assumed to abide by the occam rules.

By default, the compiler will ensure that no variables and channels are permitted to break the usage rules. Where the checks are disabled, the compiler will have to make worse assumptions about the behavior of the program, and may generate slower code.

15.2.11 Memory layout

By default, the toolset arranges memory as shown in figure 15.1.

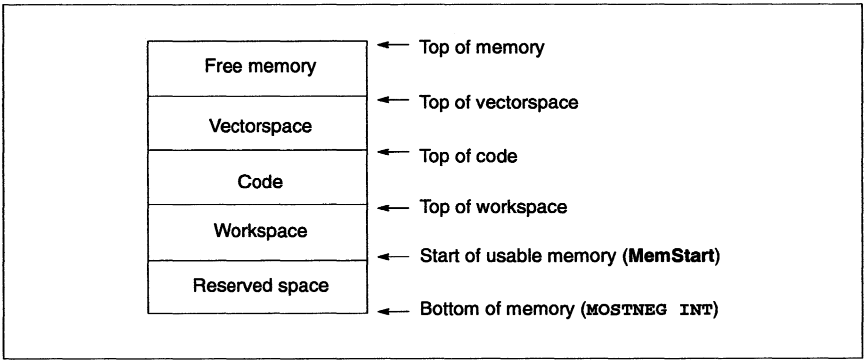


Figure 15.1 Default memory layout

When the collector `S` option is used for single processor programs, the collector allocates another buffer below the workspace; see section 15.4.6.

The configuration attributes `order.code`, `order.ws`, and `order.vs` may be used to override this default ordering. By default; the reserved space is simply that memory up to **MemStart**. However the configurer's `reserved` attribute may be used to override this. `location` attributes may also be used to override this memory layout; see section 15.4.5.

15.2.12 When there is not enough on-chip RAM

When a program's workspace is substantially larger than the on-chip RAM, it may well be true that most of the time the stack is working off-chip. In this case, it might be more useful to move the code on-chip, particularly time-critical code sections, and leave all the workspace off-chip; see sections 15.4.3 and 15.4.5.

15.3 Obtaining information

Various tools provide information which can be useful when improving performance.

- Compiler information.

  The compiler `I` command line switch displays information about the workspace and vectorspace requirements of each externally visible procedure or function. It also displays the number of bytes of code in the module.

The compiler `P` command line switch can be used to supply the name of a text file which the compiler produces known as the *map file*. This lists the layout of stack memory for each routine in the program, and the layout of the code for each routine. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- Linker information.

  The linker `MO` command line switch can be used to produce a text file which indicates how the code has been linked together. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- Collector information.

  The collector `P` command line switch can be used to produce a text file which indicates the memory layout of each processor in the network. It also indicates the processor connectivity. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- The lister tool.

  The `ilist` program can examine any data file which is created by the toolset, and display a decoded form of its contents. This may be useful if extra information is required which is not available by the previous methods.

- The mapper tool.

  The `imap` program takes the map files created by the compiler, linker, and collector, and combines the information into a single text file which lists the whole program's memory layout for each processor.

15.4 Command line switches

There are many different switches and commands which can be used with the compiler, linker, collector, and configurer, in order to modify program execution speed.

15.4.1 Compiler command line switches

The following compiler command line switches can affect program performance. Note that many options can be specified in the source code by inserting a `#OPTION` statement as the first line of the program.

A typical benchmark program would be compiled with options `H`, `NA`, `U`, and `Y`, and maybe `A` and `V` too.

`H` HALT error mode. (This is the default)

See the discussion on error modes in section 15.2.7.

S STOP error mode.

See the discussion on error modes in section 15.2.7.

X UNIVERSAL error mode.

See the discussion on error modes in section 15.2.7.

K Disable range checking.

This removes any code whose sole purpose is to check for array bounds violations. Any such check which can be performed at compile time (e.g. because the index is a constant) will still be performed. Note that the K option will also disable run-time alias checks.

U Disable run-time error checking.

This removes any code whose sole purpose is to detect errors (except ASSERT - see below). For example, code to check that the number of replications of a replicator is not negative will be omitted. It does *not* mean that errors are 'allowed'. Some errors may still be detected because the fastest code includes error checking. (E.g. adc is used to add a constant number; this instruction performs overflow checking). Note that this option is stronger than the K option, in that it does everything that the K option does, and more, so it is not necessary to specify both on the command line. Also see the discussion on error modes in section 15.2.7.

NA Disable run-time ASSERT checks.

The ASSERT predefine can be used to provide security checks. If a check can be performed at compile-time, it will be. Otherwise code is inserted to perform the check at run-time. This option disables the run-time checks.

N Disable Usage checking.

See the discussion on usage checking in section 15.2.10.

A Disable Alias checking.

See the discussion on alias checking in section 15.2.9. Note that the K option will also disable run-time alias checks.

V Disable vectorspace.

See the discussion on separate vectorspace in section 15.2.8.

Y Disable library calls for channel i/o.

See the discussion on interactive debugging in section 15.2.5. Also see the discussion on virtual routing in section 15.2.6.

P Produce map file.

This option can be used to specify the name of a text file (the 'map' file) which is created by the compiler to provide information about memory layout. This is used by the **imap** tool.

15.4.2 Linker command line switches

Use the linker H, S, or X flags to link in the correct error mode. If interactive debugging and virtual routing are not required, disable library i/o by using the Y flag.

The MO option can be used to specify the name of a text file (the 'module' file) which is created by the linker to provide information about the linkage. This is used by the **imap** tool.

15.4.3 Linker directives

The linker allows a programmer to control the relative ordering of different modules in the linked object file. The output file will still be a single consecutive chunk of code, but the relative order of subroutines can be controlled. Primarily this is done by rearranging the order in which the files are listed on the command line. The linker does, however, provide finer control than this if required.

The linker inserts all separately compiled units into the output code file in the same order as they are encountered on the command line. The first module will be loaded at a lower address, that is, nearer **MOSTNEG INT**. It then adds library modules as necessary. It chooses the entry name of the first separately compiled module to be the entry point of the whole file; normally the top level module is listed first. If you re-order the files on the command line, you must provide a **#mainentry** command to the linker (or use the **ME** command line option) to tell the linker the name of the main entry point of the program.

#section directives in the linker's input file provide finer control. By default, the occam 2.1 compiler places all code in any compilation module into a code section named **text%base**. This may be overridden by use of the compiler's **#PRAGMA LINKAGE**. If this pragma is specified, the code section is named **pri%text%base**. If the pragma is followed by a string (in double quotes), that name is used for the code section.

The linker links all code modules in any particular named section in arbitrary order, and then concatenates the sections. However, by naming different sections, a programmer can control the overall order. Normally, the linker places the section named **pri%text%base** at the beginning of the code, (i.e. nearer **MOSTNEG INT**), followed by **text%base**, followed by any other code sections. If the programmer supplies *any* **#section** directives in the linker's input file, the default is ignored. Instead, the linker places the first named section first, followed by the next named section, etc. Any sections which were not explicitly named are placed at the end. (Note that the **#section** directive should be followed by the section name *without* enclosing quotes). The module file created by the linker MO option can be examined to confirm the relative placement of sections.

Note that floating point support libraries used on T4-series transputers are automatically placed into section **pri%text%base**, so that they are more likely to be placed onto on-chip RAM.

15.4.4 Configurer command line switches

Interactive debugging should be disabled using the `Y` option, for optimum performance. This also means that a smaller start-up routine can be used, which enables more of a user's program to fit into on-chip RAM.

Similarly, virtual routing should be disabled by using the `NV` option if appropriate.

15.4.5 Configuration language attributes for optimizing memory

As described in the discussion on memory layout in section 15.2.11, by default workspace is placed at the lowest address, followed by code, followed by vectorspace. The configurer provides attributes which can be set to override this default.

Ordering attributes

The `order.code` attribute of a `PROCESSOR` can be set to an integer value, in the `MAPPING` section of the configuration source. The default value is 0. If this attribute is set to a value less than 0, the code will be placed at a lower address than the workspace. Similarly, the `order.vs` attribute can be set to a negative value to indicate that it should be placed at a lower address than the workspace. The relative values of `order.code`, `order.ws`, and `order.vs` indicate which should be placed at a lower address.

The collector's map file (produced by the `p` command line option) can be inspected to see the effect of these switches.

Note that this facility must be explicitly enabled by the configurer `RE` option, because the `idebug` debugger cannot be used if the memory layout has been altered.

Location attributes

Attributes `location.code`, `location.ws`, and `location.vs` can be used to explicitly specify where the code, workspace, or vectorspace of a program should be placed. They are set to a machine address. This address must not be within the address range used by the configurer for its own use; namely that area above `reserved BYTES` from `MOSTNEG INT`, and below `memsize BYTES` from `MOSTNEG INT`.

Reserved attribute

The `reserved` attribute can be used to tell the configurer not to use the memory immediately above `MOSTNEG INT`; i.e. the lowest addresses in memory. It should be set to the number of `BYTES` above `MOSTNEG INT` which are to be reserved. The configurer will then leave this area free for use by the programmer; either by using the `location` attributes, or by `PLACE`ing data there directly.

15.4.6 Collector command line switches

`M` Specify memory size.

When collecting a program for a single transputer, this option creates a bootable which does not examine the environment variable IBOARDSIZE at run-time. Instead, it uses the value supplied to the collector. This will reduce the memory requirements of the start-up code.

P Specify memory map file.

This option creates a map file describing the memory map of each processor, and their connectivity. This is used by the imap tool.

S Specify stack size.

This option can be used for single processor programs to gain access to a reserved block of memory at the bottom of the address space. (In the C toolset it is used to place the C program's stack onto the on-chip RAM, hence the name of the option).

A single processor program normally requires the following formal parameter list:

```
#INCLUDE "hostio.inc"
PROC myprog (CHAN OF SP fs, ts, []INT freespace)
```

This can be modified as follows:

```
#INCLUDE "hostio.inc"
PROC myprog (CHAN OF SP fs, ts, []INT freespace,
             []INT buffer)
```

The collector will allocate a buffer at the bottom of memory, and pass it as the extra parameter. If the collector's S option is not specified, this array will be of length zero. The buffer can be used to hold an array of data which is required to be accessed quickly; see the discussion about memory layout in section 15.2.11.

15.5 Compiler optimizations

The compiler already performs some minor optimizations:

- Constant folding

  Expressions such as 27 + 33 are *folded* into 60.

- Unused variable elimination

  Any variables which are never used are not allocated stack space.

- Basic dead code elimination

  Branches of IFs which can be determined at compile-time to always be FALSE are ignored.

- Constant tables

  The transputer instruction set creates large constants by repeated use of a *pfix* instruction. This can sometimes mean that loading large constant values can be slow. However, the compiler recognizes such constants, and 'caches' them into a constant table, which is accessed quickly via a small offset from a local pointer.

- Workspace allocation

  The compiler estimates the usage frequency of each variable which is used in a procedure or function, and allocates the variables in memory so that the most used variables are stored at small offsets in workspace, so as to minimize the overall execution cost.

- Merging of constant arrays

  Constant arrays and strings which appear more than once in a single compilation unit are merged into a single array, and thus only appear once in the code.

- Elimination of temporaries

  The compiler knows that different variables always refer to different data items, and can reduce the number of temporaries required in multiple assignment, etc.

15.6 Source code optimizations

The reader should be aware that many of the following source code optimizations are implementation dependent, and may actually result in a performance degradation in a different implementation of the occam language.

15.6.1 Compiler workspace layout

The current compiler allocates workspace as a falling stack. Hence the workspace for a nested procedure or function will be allocated at a lower address than that of the enclosing subroutine.

Workspace for parallel processes are allocated below the workspace of the parent. The first member of the `PAR` list (or the lowest replicator value of a replicated `PAR`) is allocated workspace immediately below the parent, the next immediately below that, etc. Thus the *last* process will have the lowest workspace address, and hence is most likely to be placed on-chip.

Suppose we have three procedures `a()`, `b()` and `c()`. Then

```
SEQ
  ...  body of parent
  PAR
    a ()
    b ()
    c ()
```

is allocated as shown in figure 15.2.

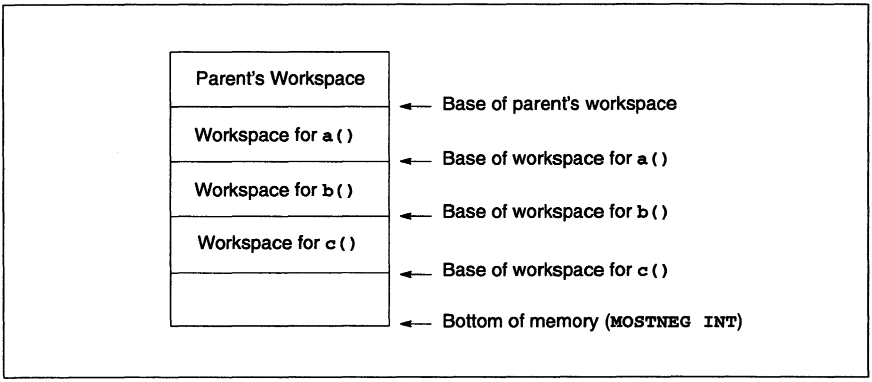

Parent's Workspace
Workspace for a()
Workspace for b()
Workspace for c()
Base of parent's workspace
Base of workspace for a()
Base of workspace for b()
Base of workspace for c()
Bottom of memory (MOSTNEG INT)

Figure 15.2 Workspace layout

This arrangement also holds for PRI PAR. Hence if it is important to get the workspace of a high priority process on-chip, where the low priority process has a large workspace, the following can be used:

```
PAR
  ...  low priority process
  PRI PAR
    ...  high priority process
    SKIP
```

15.6.2 Compiler code layout

The compiler writes code for a single compilation into a single object file. PROC and FUNCTION bodies are written in the *reverse* order of the *end* of their definition; thus if you read the source backwards, the routines are inserted into the object file in the order in which you find their terminating colons (:). This means that all calls are *forwards*, and that calls to a routine do not have to jump over the body of that routine; these considerations help make the call instruction smaller.

However, it tends to mean that nested subroutines are placed at higher addresses, which can push them out of on-chip RAM. It may be useful to make critical inner subroutines into separately compiled units, and use the linker to place that at a lower address; see section 15.4.3.

15.6.3 Abbreviations

Abbreviations are a powerful feature of the occam language. They can be used to bring non-local variables down into local scope, thus removing the need to chain through the procedure call stack, and speeding up access. They can also speed up execution by removing range check instructions. Where appropriate, VAL abbreviations should be used; for scalar values this creates a local copy of a variable rather than a pointer to it.

When performance is the main aim, abbreviations should not be used if the value is only used once or twice. Similarly, if an expression is simple, it may be faster to re-evaluate the expression, rather than to read another value from memory, especially if the workspace does not fit in on-chip RAM.

Removing range-checking code

By abbreviating sub-vectors of larger vectors and using constants to index into the sub-vector, the compiler will generate range-checking code for the abbreviation, but will not need to generate range-checking code for accesses to the sub-vector.

As an example of an abbreviation removing range check instructions, here are two versions of the same procedure. Part of the ray-tracer, this procedure is initializing fields in a new node to be added into a tree. The identifier `nodePtr` points to the start of the node. The second version uses an abbreviation, generates no range checking code (apart from initial generation of the abbreviation) generates shorter code sequences for each assignment, and executes more quickly.

```
PROC initNode ( VAL INT nodePtr )     -- version 1
  SEQ
    tree [ nodePtr + n.reflect] := nil
    tree [ nodePtr + n.refract] := nil
    tree [ nodePtr +    n.next] := nil
    tree [ nodePtr +  n.object] := nil
:

PROC initNode ( VAL INT nodePtr )     -- version 2
  node IS [ tree FROM nodePtr FOR nodeSize ] :
  SEQ
    node [ n.reflect] := nil
    node [ n.refract] := nil
    node [    n.next] := nil
    node [  n.object] := nil
:
```

Even if range-checking were switched off, the second version will execute more quickly. Without range check instructions, the statement will generate the following transputer instructions:

```
ldc    nil          -- get data to save
ldl    nodePtr      -- get pointer to base of node
ldl    static       -- get static chain
ldnlp  tree         -- generate pointer to tree ( in outer scope)
wsub                -- generate pointer to tree [ nodeptr]
stnl   n.refract    -- and store to tree [ nodePtr + n.refract]
```

whereas the second version will generate the following, appreciably shorter and faster fragment of code:

```
ldc   nil         -- get data to save
ldl   node        -- load abbreviation
stnl  n.refract   -- and store
```

Of course there is an initial overhead to generate the abbreviation, but this is rapidly swamped by the subsequent savings.

Loop unrolling

Using abbreviations in conjunction with loop unrolling by hand can speed up execution considerably. Take the following piece of occam , a simple vector addition:

```
SEQ i = 0 FOR 20000
  a[i] := b[i] + c[i]
```

The transputer loops in about a microsecond, but adds in about 50 nanoseconds. Therefore to increase performance we must increase the number of adds per loop:

```
VAL bigLoops IS 2000 / 16 :
VAL leftOver IS 2000 - (bigLoops TIMES 16) :
SEQ
  SEQ i = 0 FOR bigLoops
    VAL base IS i TIMES 16 :
    aSlice     IS [ a FROM base FOR 16 ] :
    VAL bSlice IS [ b FROM base FOR 16 ] :
    VAL cSlice IS [ c FROM base FOR 16 ] :
    SEQ
      aSlice  [0] := bSlice [0] + cSlice [0]
      aSlice  [1] := bSlice [1] + cSlice [1]
      aSlice  [2] := bSlice [2] + cSlice [2]
                 ............
      aSlice [14] := bSlice[14] + cSlice[14]
      aSlice [15] := bSlice[15] + cSlice[15]
  SEQ i = 2000 - leftOver FOR leftOver
    a[i] := b[i] + c[i]
```

Obviously, loops can be opened out in any language, on any processor, and performance will tend be improved at the expense of increased code size. However, opening loops out in slices of 16 has a knock-on effect on the transputer, as optimal code **with no prefix instructions** is generated for each addition statement. Compare the code generated for the two statements:

```
                                    ldl  i
                                    ldl  b
                                    wsub
                                    ldnl 0
                                    ldl  i
a[i] := b[i] + c[i]                 ldl  c
                                    wsub
                                    ldnl 0
                                    add
                                    ldl  a
                                    ldl  i
                                    wsub
                                    stnl 0
```

```
                                             ldl  bSlice
                                             ldnl 15
                                             ldl  cSlice
  aSlice[15] := bSlice[15] + cSlice[15]      ldnl 15
                                             add
                                             ldl  aSlice
                                             stnl 15
```

The second piece of code is just over half the size of the first and the number of loop end *lend* instructions executed is reduced by a factor of 16.

15.6.4 Vector space

Use the *allocations:*

`PLACE` *name* `IN WORKSPACE:` and `PLACE` *name* `IN VECSPACE:`

as described in the discussion on separate vector space in section 15.2.8.

Suppose we wish to clear a large block of memory, such as a clear screen operation. It may be worthwhile using an array which is placed in on-chip RAM as the source of a block move:

```
PROC ClearScreen(VAL BYTE pattern)
  -- the screen is declared as [512][512]BYTE screen :
  [SIZE screen[0]]BYTE fastvec :  -- this is in on-chip RAM
  PLACE fastvec IN WORKSPACE :
  SEQ
    initBYTEvec(fastvec, pattern) -- fast BYTE initializer
    SEQ y = 0 FOR SIZE screen
      screen[y] := fastvec        -- use a block move
:
```

This fires off 512 block move instructions, each of 512 bytes. Since the block move is reading from on-chip memory, and writing to off-chip memory, it will proceed more quickly than

```
PROC ClearScreen(VAL BYTE pattern)
  -- the screen is declared as [512][512]BYTE screen :
  [(SIZE screen) * (SIZE screen[0])]BYTE bytescreen :
  initBYTEvec(bytescreen, pattern) -- fast BYTE initializer
:
```

where all data accesses are to off-chip memory. The time saved during the block moves outweighs the cost of setting up the parameters to the block moves, and of the initial `initBYTEvec`. See section 15.6.7 for more about block moves and for the source of `initBYTEvec`.

15.6.5 Beware the PLACE statement

A common mistake in trying to make occam go faster is to physically place data on-chip, using a `PLACE AT` statement. This does the right thing - the compiler will physically place the variable on-chip, but the variable will be outside local workspace. Therefore to access the variable, its physical address must be generated, and an indirection performed to load the contents of the address. For example, declaring a variable at word address 30 above MOSTNEG INT, and setting its value to 3 :–

```
INT a :
PLACE a AT 30 : -- 30th word above MOSTNEG INT
a := 3

ldc 3
mint
ldnlp 30
stnl 0
```

This code sequence takes 7 cycles (350 ns on a T800-20). Were a a local variable, the code sequence would take only 3 cycles (150 ns) if the workspace were on-chip, and would be:

```
ldc 3
stl a
```

In any case it is dangerous to place variables directly into on-chip RAM, because unless the on-chip RAM has been reserved in some other way, the explicit allocation to on-chip RAM will clash with some other code or data which is already there.

The key to making variable accesses go faster is to **keep the workspace on-chip**. Then if it is necessary for a vector to be on-chip, it can be declared in local scope.

15.6.6 Abbreviating PLACED objects

In some circumstances PLACED objects must be used, for example to talk to some external hardware such as a UART. In this case, it is often more efficient to create a local copy of the address by using an abbreviation, rather than referring repeatedly to the original object:

```
PORT OF INT uart :
PLACE uart AT #12345 :

PORT OF INT uart IS uart :  -- forces a local copy of the
address
SEQ
  ...  use 'uart'
```

15.6.7 Block move

The transputer block instruction *move* is directly supported by the occam language. The array assignment statement:

```
[65536] BYTE bigVec, otherVec :
[bigVec FROM 0 FOR 65536] := [otherVec FROM 0 FOR 65536]
```

compiles down to only four instructions:

```
ldl bigVec       -- assuming the vectors are abbreviated
ldl otherVec     -- locally
ldc 65536        -- this will be prefixed of course
move
```

A very fast array initializer can be written using block moves:

```
PROC initBYTEvec ([]BYTE vec, VAL BYTE pattern)
  INT dest, transfer :
  SEQ
    transfer := 1
    dest     := transfer
    vec [0]  := pattern
    WHILE dest < (SIZE vec)
      SEQ
        IF
          (dest + transfer) > (SIZE vec)
            transfer := (SIZE vec) - dest
          TRUE
            SKIP
        [vec FROM dest FOR transfer] := [vec FROM 0 FOR transfer]
        dest     := dest     + transfer
        transfer := transfer + transfer
:
```

This performs a series of assignments of increasing length, initializing the first byte of the array, then the next 2, then the next 4, 8, 16 etc., until it finishes the array.

15.6.8 Use TIMES

The transputer has a fast (but unchecked) multiply instruction, which is accessed with the occam operator TIMES. A checked integer multiply on the IMS T414-20 takes over a microsecond, whereas using TIMES this will take as many processor cycles as there are significant bits in the right-hand operand, plus 2 cycles overhead.

a TIMES 4 takes only 6 cycles (300 ns). Therefore, when multiplying integers by small constants, use TIMES. Current transputers have a modified version of TIMES which optimally multiplies small negative integers.

The compiler uses this faster, unchecked, version of multiply for normal multiply operations if run-time error checking is disabled by means of the U command line switch.

15.6.9 Retyping – accelerating byte manipulation

Under certain circumstances retyping can be used to speed up byte manipulation. If it is necessary to frequently extract byte fields from a word, then retyping the word to a BYTE array is faster than shifting and masking. For example:

```
INT word :
[4] BYTE bWord RETYPES word :
SEQ
  ...  use bWord[0], bWord[1], bWord[2], bWord[3]
```

To access bits 16..23 in word, simply reference bWord[2], which will generate:

```
ldlp  bWord  -- load base of bWord
adc   2      -- select byte 2
lb           -- and load it
```

To perform byte operations on large arrays it is worthwhile moving portions of the array to a local (on-chip) array; this is because a block move transfers words and is therefore much faster than accessing individual bytes from an off-chip array. For example:

```
[1024] INT vector :
[]BYTE bytevector RETYPES vector :

[16] BYTE local :
PLACE local IN WORKSPACE :
INT base :
SEQ
  base := 0
  SEQ i = 0 FOR 64
    SEQ
      local := [bytevector FROM base FOR 16]
      base := base + 16
      SEQ i = 0 FOR 16
        SEQ
          ...  use local[i] to access each byte
```

15.6.10 Scoping of variables

The compiler estimates the run-time count of the number of times each variable in a subroutine is accessed, using a heuristic which allows for repetition of loops, etc. It uses this information to place the most frequently used variables at the smallest workspace addresses; hence these variables can be accessed via smaller (and faster) instructions. The order of variable declarations has no major effect, unlike the situation in the IMS D705B occam toolset; and the use of separate vectorspace makes the problem less acute anyway.

The scope of variables should be as local as possible. The compiler uses the lexical scoping of variables to determine which variables are live (in use) at the same time, and which can be overlaid over each other. Hence localized scoping of variables can also reduce the total workspace requirement, thus helping to fit the total workspace into on-chip RAM.

BYTE and BOOL scalar variables are initialized at declaration by the compiler, to enable quick access as a local variable. Therefore it is not a good idea to declare them inside inner loops.

Note: Unfortunately this is not good programming practice. Declaring items at the scope within which they are required is more secure, preventing accidental modification and other programming errors.

15.6.11 Use the whole language

There are features of occam which are particularly suited to certain types of problems.

For example, when comparing an expression against a list of distinct constants, use a CASE statement rather than an IF. A compiler will attempt to make a CASE construct as fast as possible, assuming that all the values are equally possible, and may use a combination of techniques to select the correct branch. This compiler uses a combination of jump tables, binary searches, and explicit tests, depending on the relative values and density of the target values (i.e. whether there are any gaps).

An IF construct must be executed sequentially, evaluating each of the guards in turn. The first guard which is TRUE will be executed. Thus branches which are likely to be chosen frequently should be listed at the start of the IF.

Note that a combination of these two constructs may be the best solution where one value is particularly common, but where there may be many other possibilities:

```
VAL temp IS complicated.expression :
IF
  temp = frequent.value
    ...  process frequent.value
  TRUE
    CASE temp
      infrequent.value0
        ...  process infrequent.value0
      infrequent.value1
        ...  process infrequent.value1
      infrequent.value2
        ...  process infrequent.value2
      ...  etc
```

Note that replicated IFs are particularly suited to search-type lookups and comparisons:

```
BOOL FUNCTION equal.string(VAL []BYTE a, b)
  -- This returns TRUE if a = b
  BOOL result :
  VALOF
    IF
      (SIZE a) <> (SIZE b)
        result := FALSE
      IF i = 0 FOR SIZE a
        a[i] <> b[i]
          result := FALSE
      TRUE
        result := TRUE
    RESULT result
:
```

15.6.12 INLINE procedures and functions

This compiler allows you to write the keyword INLINE immediately before the keyword PROC or FUNCTION of a procedure or function declaration. The effect is that any call of that subroutine is expanded out as though the body were written in-line at the call site. This can be used to greatly improve program readability with no loss of performance. More importantly, this can improve performance by removing the overhead of the procedure or function call. It also allows the compiler to compile the body of the routine knowing the actual parameters, which provides further opportunities for optimization.

The programmer should be aware that inlining normally increases code size, and can cause problems because the calling procedure is then enlarged.

The current implementation does not permit the definition of the procedure or function which is to be inlined to exist in a different separately compiled unit to the caller. Instead, the declaration should be written in an include file, and #INCLUDEd by every source file which calls the routine.

15.6.13 Access to non–local variables

Non-local variables (i.e. those which are declared in an outer procedure or function) are accessed via a *static link*. This will require one memory access for each level of nesting, every time that variable is accessed. It is possible to avoid doing this repeatedly, for example when a variable is used inside a loop, by creating a local abbreviation to that variable; this will create a pointer in the local workspace, and this local copy can be used inside the loop.

15.6.14 Access to formal parameters

All variable non-VAL parameters, and all VAL parameters which are either arrays, or longer than a word, are accessed via a pointer. If a parameter is accessed frequently, it may be worthwhile saving such a variable in a local variable:

```
PROC cumulative.sum(INT sum, VAL []INT array)
  INT local.sum :
  SEQ
    local.sum := sum    -- copy into a local variable
    SEQ i = 0 FOR SIZE array
      local.sum := local.sum + array[i]
    sum := local.sum    -- write back into the real variable
:
```

15.6.15 Pre–evaluate expressions

This technique is applicable to all programming languages. Any calculation which is to be performed repeatedly should be removed from any inner loops. If the calculation is relatively simple, it can be pre-evaluated by hand. Alternatively, at the beginning of the program a table can be initialized so that the values can be accessed quickly later.

As a trivial example, suppose a program requires frequent access to n^3 where n is always less than 100.

```
PROC init.cubes([]INT cubes)
  SEQ i = 0 FOR SIZE cubes
    cubes[i] := (i * i) * i
:
[100]INT cubes :
SEQ
  init.cubes(cubes)
  ...  other initialisation
  WHILE test  -- this is the 'inner loop'
    ...  use "cube[x]" instead of "(x * x) * x" here
```

15.6.16 Conditional expressions

Remember that by definition `INT TRUE` evaluates to 1 and `INT FALSE` to 0. This can be used to transform the following type of example:

```
SEQ
  IF               -- this is slower
    test
      x := 1
    TRUE
      x := 0
```

Using the `INT` type conversion operator this becomes:

```
x := INT test  -- this is quicker
```

Note that some programmers consider the second form to be less readable, so the first could be left as a comment as is probably better for code that is not performance critical.

15.6.17 Array subscripts

Array subscripts of the form `a[c]` (where `c` is a constant) are evaluated most efficiently. However, if no range checks are required, and run-time error checking is disabled, `a[e + c]` (where `e` is any expression), is evaluated as quickly as `a[e]`, as is `a[c + e]` and `a[e - c]`.

15.6.18 `INT16`s

`INT16` values are not handled very efficiently on current 32-bit transputers. They should be converted to `INT` while being processed, and converted back to `INT16` to be stored, if they are really required.

If a mixed system of 16-bit and 32-bit transputers is being used, it may be more efficient to use `INT32`s as the portable communication values, since `INT32` values on a 16-bit processor are generally handled more efficiently than INT16 values on a 32-bit processor. However, using INT32s will require twice as much data to be communicated and stored. The memory requirements may be particularly important on a 16-bit processor which only has 64 kbytes of addressable memory.

15.6.19 ALTs

Large, multi-way ALTs are relatively slow, since their time cost is proportional to the number of channels in the ALT. A technique known as fan-in can be used to enhance their speed.

Instead of, say, a 100-way ALT, it would be faster to use ten processes consisting of 10-way ALTs collecting the input, and passing that information to another process with another 10-way ALT. Each communication through the ALT then is processed by two 10-way ALTs instead of one 100-way ALT, and will be faster. Care must be taken, however, because using this model will change the synchronization properties of the program.

15.6.20 Use of ASSERT()

This compiler implements a predefined procedure ASSERT(VAL BOOL test). If test is FALSE, and can be detected at compile time, this causes a compile-time error. If test can only be evaluated at run-time, the compiler will insert code to check that test is TRUE.

This can be used for various debugging tests, and to document and check various assumptions which have been made in the source code. If required, run-time ASSERT checks can be removed by using the compiler NA command line option.

15.6.21 Transputer scheduler

This compiler implements a predefined procedure RESCHEDULE(). This will place the current process to the back of the active process queue, and will work in either priority. RESCHEDULE should be used rather than relying on the implications of the transputer scheduling model. In some implementations of occam, the following code reschedules the current process, but a compiler is quite at liberty to optimize it out completely.

```
PAR
  SKIP
  SKIP
```

15.7 Summary

15.7.1 Optimizing for code size

Most of the optimizations described in this chapter will optimize for both space and time, but note:

- If an `INLINE` routine is called more than once, its body will be expanded multiple times.
- The use of separate vectorspace may cost code space because an extra parameter must be passed to every routine.

On the other hand, careful use of abbreviations can also reduce code size.

15.7.2 Removing run–time checks

Remember that run-time checks are not as costly as is sometimes thought.

The compiler's `K` switch removes any checks concerning array bounds violations. The `NA` switch removes all run-time checks of `ASSERT`. The `U` switch removes all code whose sole purpose is to detect errors (e.g. integer conversions, etc.). The `Y` switch disables library calls for channel i/o and will speed up communication.

So to compile a program with no error checking, use:

```
oc myprog -h -na -u -y
```

15.7.3 Placing arrays in on–chip RAM

If the access time of a particular array is critical, and the workspace of a program fits into on-chip RAM, it will be useful to move that array into on-chip RAM instead of vectorspace, by inserting the following declaration after the array's declaration:

```
PLACE array.name IN WORKSPACE :
```

If the workspace does not already fit into on-chip RAM, it might be worth putting the vectorspace at the bottom of memory, instead of the program's workspace. Turn vectorspace off by default, by using the compiler's `V` option, but place the critical array into vectorspace explicitly:

```
PLACE array.name IN VECSPACE :
```

The `order.vs` configurer attribute can then be used to move the vectorspace to the bottom of memory. This attribute must be set in the `MAPPING` section of the program; see section 15.4.5.

```
MAPPING
  SET processor (order.vs := -1)
:
```

An alternative method which can be used for single processor programs which are not configured, is to use the collector's s switch (see section 15.4.6). The program should be declared with an extra parameter as follows:

```
#INCLUDE "hostio.inc"
PROC myprog(CHAN OF SP fs, ts, []INT mem, []INT fastmem)
  ...
:
```

The fastmem array is placed onto the on-chip RAM by the collector, and is allocated so that the space is not used for anything else. Hence this array can then be used for buffers, etc., which are required to be accessed quickly. For example, it could be used as the initializer buffer for the ClearScreen subroutine described in section 15.6.4. The collector should be invoked as follows:

```
icollect myprog.lku -t -s 100
```

This will allocate 100 words at the bottom of memory, which are passed in as the parameter fastmem, and can be accessed more quickly than the rest of memory.

15.7.4 Placing code in on–chip RAM

Turn the critical subroutine into a separately compiled routine by passing all 'global' variables into it as parameters.

Add the following directive to the beginning of the subroutine:

```
#PRAGMA LINKAGE
```

This informs the linker to place the code for this subroutine in front of the rest of the code.

If the workspace requirement of the program is small and fits in on-chip RAM anyway, some of the code will be placed on-chip too. Since the linker has placed the critical routine at the start of the code section, this routine will be placed in on-chip RAM.

If the workspace requirement is large, it may be better to move the entire workspace off-chip, so that the code can be placed in on-chip RAM. This is done by setting the order.ws configurer attribute in the MAPPING section of the configuration program, which forces the workspace to reside at a higher address than the code; see section 15.4.5.

```
MAPPING
  SET processor (order.ws := 10)
:
```

15.7.5 Building benchmarks

Compile the program in HALT error mode, and turn off all error checks with the U switch (see section 15.4.1), and the NA switch.

Disable library calls for channel i/o with the Y compiler switch. Use the linker and configurer Y switches too; see section 15.2.5.

Also, disable virtual routing with the configurer's NV option; see section 15.2.6.

Single processor programs may benefit from using the collector M option to specify the memory size in advance, so that a simpler bootstrap may be used; see section 15.4.6.

Experiment whether disabling vectorspace has any useful effect; this may be true if the workspace requirement is small anyway, as it commonly is with benchmarks; see section 15.2.8.

Take particular care to ensure that the workspace is placed in on-chip RAM. Where possible, use linker section ordering to ensure that the inner loop subroutines are also placed on-chip.

15.8 Maximizing multiprocessor performance

The following sections will describe how to obtain more performance from an array of transputers. However, only very general guidelines can be offered. Maximizing multiprocessor performance is still an area of active research, and any solution will tend to be specific to the problem at hand.

15.8.1 Maximizing link performance

The transputer links are autonomous DMA engines, capable of transferring data bidirectionally at up to 20 Mbits/sec. They are capable of these data rates without seriously degrading the performance of the processor. To achieve maximum link throughput from a multi transputer system the links and the processor should all be kept as busy as possible.

Decoupling communication and computation

To avoid the links waiting on the processor or the processor waiting on the links, **link communication should be decoupled from computation.**

For example, the following program is part of a pipeline, inputting data, applying a transformation to each data item, then outputting the transformed data:

```
PROC transform (CHAN OF protocol in, out)
  [dataSize] INT data :
  WHILE TRUE
    SEQ
      in  ? data
      applyTransform ( data )
      out ! data
:
```

If the channels in and out are transputer links, then the performance of the pipeline will be degraded. The SEQ construct is forcing the transputer to perform only one action at a time; it is either inputting, computing or outputting; it could be doing all three at once. Embedding the transformer between a pair of buffers will improve performance considerably:

```
PAR
  buffer ( in, a )
  transform ( a, b )
  buffer ( b, out )
```

The buffers are decoupling devices, allowing the processor to perform computation on one set of data, whilst concurrently inputting a new set, and outputting the previous set.

In this example the buffer processes will simply input data then output it. There is a transfer of data here which can be avoided, as all the data can be passed by reference:

```
[dataSize] INT a, b, c :
...  proc input
...  proc transform
...  proc output
SEQ
  input (a)                   -- start-up sequence .. pull in data
  PAR
    input (b)                 -- now transform that data
    transform (a)             -- and pull in more ...
  WHILE TRUE
    SEQ                       -- and from here on
      PAR                     -- the buffers pass round-robin
        input     (c)         -- between the inputter, transformer
        transform (b)         -- and outputter
        output    (a)
      PAR
        input     (a)
        transform (c)
        output    (b)
      PAR
        input     (b)
        transform (a)
        output    (c)
```

Instead of input and output operations transferring data between the processes, the processes transfer themselves between the data, each process cycling between the vectors a, b and c as the PAR statements close down and restart.

This is a special case, a data flow architecture where all communication and processing is synchronous — there is a lock-step **in, transform, out** sequence which allows this sequential overlay of computing and communication. This is not the case in many programs, where buffer processes are required.

Priority

Correct use of priority is important for most distributed programs communicating via links. If a message is transmitted to a transputer and requires through-routing, it is essential that the transputer input the message then output it with minimum delay — another transputer somewhere in the system could be held up, waiting for the message. In such cases it is important to **run the processes which use the links at high-priority**. There will tend to be more than one process talking to links, at most eight, and the PRI PAR statement allows only one process at each priority level. It is necessary to gather together all the link communication processes, unify them into a process with a PAR statement, and run this process at high-priority.

The program from above now becomes:

```
[dataSize] INT a, b, c :
...  proc input
...  proc transform
...  proc output
SEQ
  input (a)             -- start-up sequence .. pull in data
  PRI PAR
    input     (b)       -- now transform that data (HI-PRI)
    transform (a)       -- and pull in more ...
  WHILE TRUE
    SEQ                 -- and from here on
      PRI PAR           -- the buffers pass round-robin
        PAR
          input   (c) -- between the inputter, transformer
          output  (a)
        transform (b) -- and outputter
      PRI PAR
        PAR
          input   (a)
          output  (b)
        transform (c)
      PRI PAR
        PAR
          input   (b)
          output  (c)
        transform (a)
```

15.8.2 Large link transfers

Setting up a transfer down a link takes about about a microsecond (20 processor cycles), but once that transfer is started it will proceed autonomously from the processor, consuming typically 4 processor cycles every 4 microseconds (one memory read or write cycle per 32-bit word).

Keep messages as long as possible.

For example:

```
[300] INT data :
SEQ
  out ! some.data; 300; [ data FROM 0 FOR 300]
```

is far better than

```
[300] INT data :
SEQ
  out ! some.data; 300
  SEQ i = 0 FOR 300
    out ! data [i]
```

However, long link transfers increase latency when data must be through-routed. Some optimal message length will give the best compromise between overhead on setting up transfers, and overhead on through-routing.

15.9 Dynamic load balancing and processor farms

Processor farms are a general way of distributing problems which can be decomposed into smaller independent sub-problems. Processor farms are described in Transputer Technical Note 22 (*Communication Process Computers*), which was also supplied as chapter 4 of *The Transputer Applications Notebook - Architecture and Software.*

If implemented carefully, processor farms can give linear performance in multi transputer systems — that is ten processors will perform 10 times as well as one processor. Processor farms come into their own when solving problems where the amount of computation required for any given sub-problem is not constant.

For example, in the ray tracer example one pixel may only require one traced ray to determine its color, but other pixels may require over a hundred.

Rather than give each processor say one tenth of the screen (assuming ten processors in the array), the screen is split into much smaller areas — in this case 8x8 pixels, giving a total of 4096 work packets for a 512x512 pixel screen. These are handed out piecewise to the farm. Each processor in the farm computes the colors of the pixels for that small area, and passes the pixels back, the pixel packet being an implicit request for another area of screen to be rendered.

Since work is only given to the farm on demand, load is balanced dynamically, with the whole system keeping itself as busy as possible. Buffer processes overlay data transfer with communication, reducing the communication overhead to zero, and the end-case latency of a processors farm implemented this way is far lower than in a statically load-balanced system.

The key to the processor farm is a valve process, allowing work packets into the farm only when there is an idle processor. The structure of this valve is:

```
PAR
  -- pump work unconditionally
  SEQ i = 0 FOR workPackets
    inject ! packet
  -- regulate flow of work into farm
```

```
SEQ
  idle := processors
  WHILE running
    PRI ALT
      fromFarm ? results
        idle := idle + 1
      (idle > 0) & inject ? packet
        SEQ
          tofarm ! packet
          idle := idle - 1
```

where the crucial statement is the guarded ALT,

```
(idle > 0) & inject ? packet
```

only allowing work to pass from the pumper into the farm when there is an idle processor. The ALT is prioritized to accept results.

Appendices

A Equivalent data types

This appendix may be used for constructing interfaces between sections of code in different languages, primarily for calling occam routines from C and calling C functions from occam. The information may also be useful when using channels between occam and C processes. The 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.

A.1 occam as the calling language

This section shows an example of passing parameters from occam to C. Tables of the equivalent data types used are given in sections A.1.2 and A.1.3.

A.1.1 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 cfunc1(int parm1);

#pragma IMS_nolink(cfunc1)    /* remove the gsb hidden parameter */

void cproc1(char c, int i,
            long int l, float f,
            char *cp, short int *sp,
            int *ip, long int *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,
            long int l, float f,
            char *cp, short int *sp,
            int *ip, long int *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 fragment 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
    -- set up function and procedure parameters
    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 int s, char *cp,
            short int *sp, int *ip,
            long int *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 int s, char *cp,
            short int *sp, int *ip,
            long int *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 fragment 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
    -- set up function and procedure parameters
    result := cfunc1(i)
    cproc1(c, i, s, cp, sp, ip, lp, fp, dp, array1, array2)
```

A.1.2 Parameter passing

Table A.1 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 this is indicated.

C formal parameter	occam actual parameter	
	32-bit transputer	16-bit transputer

`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 11.1.6.	See section 11.1.6.
Defined by a `typedef`	As the underlying type.	As the underlying type.

†There is no direct type equivalent in occam. Either recode the C program or pass 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.

Table A.1 Parameter passing - occam equivalent data types to pass to C

A.1.3 Return values

Table A.2 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 transputer	16-bit transputer

`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†
Defined by a `typedef`	As the underlying type.	As the underlying type.
†There is no direct type equivalent in occam. Either recode the C program or pass the parameter in another form.		

Table A.2 Parameter passing - occam equivalents of data types received from C

A.1.4 `typedef` types

C types defined by a `typedef` statement may be passed using the guidelines given above applied to the underlying type.

A.2 C as the calling language

This section shows an example of passing parameters from C to occam. Tables of the equivalent data types used are given in sections A.2.2 and A.2.3.

A.2.1 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
:
```

A C code fragment 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 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 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(vi32);

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);
```

A C code fragment 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 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(&vi32);

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);
```

A.2.2 Parameter passing

Table A.3 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 this is indicated.

occam formal parameter	C actual parameter	
	32-bit transputer	16-bit transputer
`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 11.1.6.	See section 11.1.6.

`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 128).	Pass nothing (see page 128).
Array	See section 11.1.6.	See section 11.1.6.
`RECORD`	No direct equivalent†	No direct equivalent†
Named type	As the underlying type.	As the underlying type.

†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 SGS-THOMSON specific type declared in the C header file `channel.h`.

Table A.3 Parameter passing - C equivalent data types to pass to occam

A.2.3 Return values

Table A.4 outlines the conventions that must be followed when receiving occam function return values in C.

occam function type	C function type	
	32-bit transputer	16-bit transputer
`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`
`RECORD`	No direct equivalent†	No direct equivalent†
Named type	As the underlying type.	As the underlying type.
†There is no direct type equivalent in C. Either recode the occam program or pass the parameter in another form.		

Table A.4 Parameter passing - C equivalents of data types received from occam

A.2.4 Named types

occam named types may be passed using the guidelines given above applied to the underlying type.

B Transputer code insertion

This appendix describes the facilities for inserting transputer instructions into occam programs, using the ASM construct. In-line assembly code is described in section 12.3.

B.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.

B.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 B.3.

B.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 T2/T4/T8-series transputer instructions can be found in *The transputer instruction set: a compiler writer's guide*; the ST20450 instruction set is listed in the *ST20 Instruction Set Reference Manual.*

B.2 ASM construct

The ASM construct provides the ability to insert transputer code sequences into occam programs.

B.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.

B.2.2 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*. This means that if `x` is a non-local variable, the operand used will be the variable's offset in the non-local workspace. Note that this means non-`VAL` parameters will appear as pointers; for example, `LDL x` where `x` is a non-`VAL` parameter, will return the pointer to `x`. 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.

B.2.3 Pseudo operations

The *pseudo.op* operations are listed in table B.1.

Pseudo-op	Action
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 workspace' 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.*

Figure B.1 Inline code pseudo-operations

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.

B.3 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 or the ST20450.

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
move2dnonzero
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

The following instructions are supported by the G option, but apply only to the ST20450:

causeerror
cb
cbu
cir
ciru
clockdis?
clockenb?
cs
csu
devlb
devls
devlw
devmove
devsb
devss
devsw
gintdis?
gintenb?
gtu
insertqueue
intdis
intenb
lbx
ldclock
ldprodid
ldshadow
ldtraph
ldtrapped
ls
lsx
nop
reboot?
restart
satadd
satmul
satsub
settimeslice
signal
slmul
ss
ssub
stclock
stshadow
sttraph
sttrapped
sulmul
swapqueue
swaptimer
timeslice
trapdis?
trapenb?
trapret
wait
xbword
xsword

ASM pseudo-operations are also permitted when sequential code insertion is enabled.

C Glossary

Within the definition of a term, items which appear in **bold** are special names such as register names or tool names. Items in *italics* are terms defined elsewhere in the glossary.

Alias

If two or more expressions denote the same object, then the expressions are aliases of one another.

Alias check

A program compilation check that ensures that no two names denote the same object within a given scope.

Analyse

An IMS T2/T4/T8-series transputer and ST20450 input pin which is held high to indicate that a reset is for debugging. To reset with the **Analyse** pin held high.

A network analyzer is a tool which tests the types of transputers in the network and how they are connected.

Big endian

The opposite of *little endian.*

Bootable code

Self-starting program code that can be loaded onto a transputer or transputer network down a *link* and run. Bootable code is produced by `icollect` from linked units or configuration binary files.

Bootstrap

The first piece of code which is run on the transputer before the *loader* or the application code. Depending on the transputer type the bootstrap performs some or all of the initialization of the transputer.

Capability

A text string which identifies a transputer *resource* (or resources).

Channel

A communication channel. Channels are unidirectional, point-to-point connections between processes.

Compactable code

Object code generated by the C compiler which may be compacted by the linker. Rather than assuming a worst case size for a variable length instruction, the compiler leaves information in the object file for the linker to use to determine the optimal length of the instruction.

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.

In the context of the programmable memory interface, configuration means the selection of parameters to control the memory interface. A memory configuration file called a *memfile* is created by the memory configuration tool. A *memfile* must be created for ST20450 targets and is referenced from the configuration description (see above). For other IMS T4/T8-series transputers the configuration of the memory interface is optional and if required the memory configuration file is referenced from an EPROM control file which formats applications for input to ROM.

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. It provides and maintains connections to transputer systems across a network.

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.

Data network

Used to describe the network of communication links on a transputer network.

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 compiled by `icc` are in UNIVERSAL mode, which is the mode that can be mixed with HALT and STOP code generated by other compilers.

HALT mode halts the transputer when a program error occurs. In STOP mode only the errant process is stopped immediately, allowing other processes to continue until the STOP is gradually propagated through the system. UNIVERSAL mode enables programs to run in either STOP or HALT mode.

Error signal (or error flag)

For IMS T2/T4/T8-series transputers: 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 IMS T2/T4/T8-series transputer networks. Error signals can be OR-ed together on IMS T2/T4/T8-series transputer boards to indicate that an error has occurred in one of the transputers in a network.

For ST20450 transputers see also *Trap*.

Ethernet

A *LAN* technology based on a passive coaxial cable which transmits at 10 Mbps.

Exception

The result of a hardware detected runtime violation (arithmetic overflow, misaligned access etc.)

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 IMS T4 and IMS T8-series transputers have a programmable EMI which can be configured for different types and speeds of external memory device. See also *Programmable memory interface.*

Event

On IMS T2/T4/T8-series transputers 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 or a software routing process that multiplexes and handles communications.

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`.

Include files may also be referenced from configuration descriptions and *linker indirect files* which are input to the configurer and linker respectively. These include files have the extensions `.inc` and `.lnk` by convention.

Interrupt

A fast context switch from one priority to a higher priority. Interrupts may occur after any instruction or during certain long instructions such as block moves.

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 input 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 should have the `.liu` extension.

Link

In the context of transputer hardware, the communication link between processors. IMS T9000 transputers have data-strobe links (DS-links), while all other transputer have over-sampled links (OS-links).

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.

Linker indirect file

A linker indirect file contains a list of compiled object files, or compiled library files which are to be linked into a linked object file. Linker indirect files may also contain directives to the linker.

Linkops

A software link interface, used by host interface software.

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. In serial communications the least significant data is communicated first.

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.

LoadStart

The lowest address at which code and data may be loaded.

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`.

Memfile

A text file describing an ST20450 memory configuration, as generated by the `imem450` tool.

MemStart

The lowest address above the memory reserved for processor internal use.

Mis-aligned access

An access to an object in memory which is not located on its appropriate boundary (e.g. words must be word aligned rather than half-word i.e. byte aligned).

Mostneg

The most negative integer – 0x80000000 on a 32-bit transputer or 0x8000 on a 16-bit transputer. This value is the bottom of the transputer's address space and acts as the null pointer. It is also called `MOSTNEG INT` and MinInt.

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*. 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 preamble is a small optional piece of user code, executed before the *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. Transputers support two levels of priority, known as 'high' and 'low'.

Process

Self-contained, independently executable code.

Programmable memory interface (PMI)

Most 32-bit transputers have a programmable memory interface. The IMS T450 has four memory banks which must be *configured* before using the external memory. For all other transputers, configuring the memory interface by software is optional. See also *External memory interface.*

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

On IMS T2/T4/T8-series transputers: the reset/analyse pin.

Root transputer (or root processor)

The processor in a transputer network which is connected to the host computer directly or via a *switch*, and through which the transputer network is loaded.

Server

A program running on a host computer which provides access to the file 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 an application onto the transputer network.

Soft channels

Channels declared and used within, or between processes running on a single transputer (see also *hard channels*). Soft channels are implemented in memory.

Standard error

Specifies the standard error output device, for example, the terminal screen. For details of how to modify standard error on the system, consult the operating system documentation.

Standard input

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

Specifies the standard output device, for example, the terminal screen or a disk file. For details of how to modify standard output on the system, consult the operating system documentation.

Subsystem

In transputer board architecture (on IMS T2/T4/T8-series transputers), the combination of the *Reset*, *Analyse* and *Error* signals which allows one board to control another board connected to its *subsystem* port.

Switch

A switch, such as the ST C004 link switch, allows link connections to be controllable from software.

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 Systems databook.

Trap

A mechanism provided by some processors to enable a program to handle *exceptions*, which may be used for exception handling or by a run-time kernel or operating system.

Trap handler

A piece of code (and associated storage) called when a *trap* occurs.

Usage check

A compilation check that ensures no variables are shared between parallel processes, and that enforces rules about the use of channels as unidirectional, point-to-point connections.

Vector space

The data space required for the storage of arrays within occam code (see also *workspace*).

Virtual channel

A virtual channel is either a *hard channel* which is implemented using the *virtual channel processor* or a *soft channel* implemented by software through routing kernels.

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. Also known as a network analyzer.

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.

D Bibliography

D.1 Transputers

The transputer databook (Third Edition 1992)

INMOS Limited, July 1992
INMOS document number 72 TRN 203 02

The military and space transputer databook (First Edition 1990)

INMOS Limited, July 1990
INMOS document number 72 TRN 224 00

Transputer instruction set: A compiler writer's guide

INMOS Limited
Prentice Hall 1988

The T9000 Transputer Hardware Reference Manual (First Edition 1993)

INMOS Limited 1993
INMOS document number 72 TRN 238 01

The T9000 Transputer Instruction Set Manual (First Edition 1993)

INMOS Limited 1993
INMOS document number 72 TRN 240 01

Networks, Routers & Transputers: Function, Performance and Application

Edited by: MD May, PW Thompson, PH Welch
INMOS Limited
IOS Press 1993

Transputer Hardware and systems design

JC Hinton and AL Pinder
Prentice Hall 1993

The transputer handbook

Ian Graham and Tim King
Prentice Hall 1990

D.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

D.3 occam programming

occam 2 reference manual

INMOS Limited
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

Parallel Programs for the Transputer

Ronald S Cok
Prentice Hall 1991

D.4 Technical notes

The transputer applications notebook:
Architecture and software (First Edition 1989)

INMOS Limited, May 1989
INMOS document number 72 TRN 206 00

The transputer applications notebook:
Systems and performance (First Edition 1989)

INMOS Limited, 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

T9000 Toolset Hardware Configuration Manual

INMOS Limited, 1994
INMOS document number 72 TDS 427 00

HTRAM specification

INMOS Limited, 1993
INMOS document number *42 1567 01*

D.5 Development systems

The transputer development and ιq systems databook (Second Edition 1991)

INMOS Limited, 1991
INMOS Document Number 72 TRN 219 01

IMS B300 TCPlink hardware manual

INMOS Limited, June 1991
INMOS Document Number 72 TRN 229 01

D.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